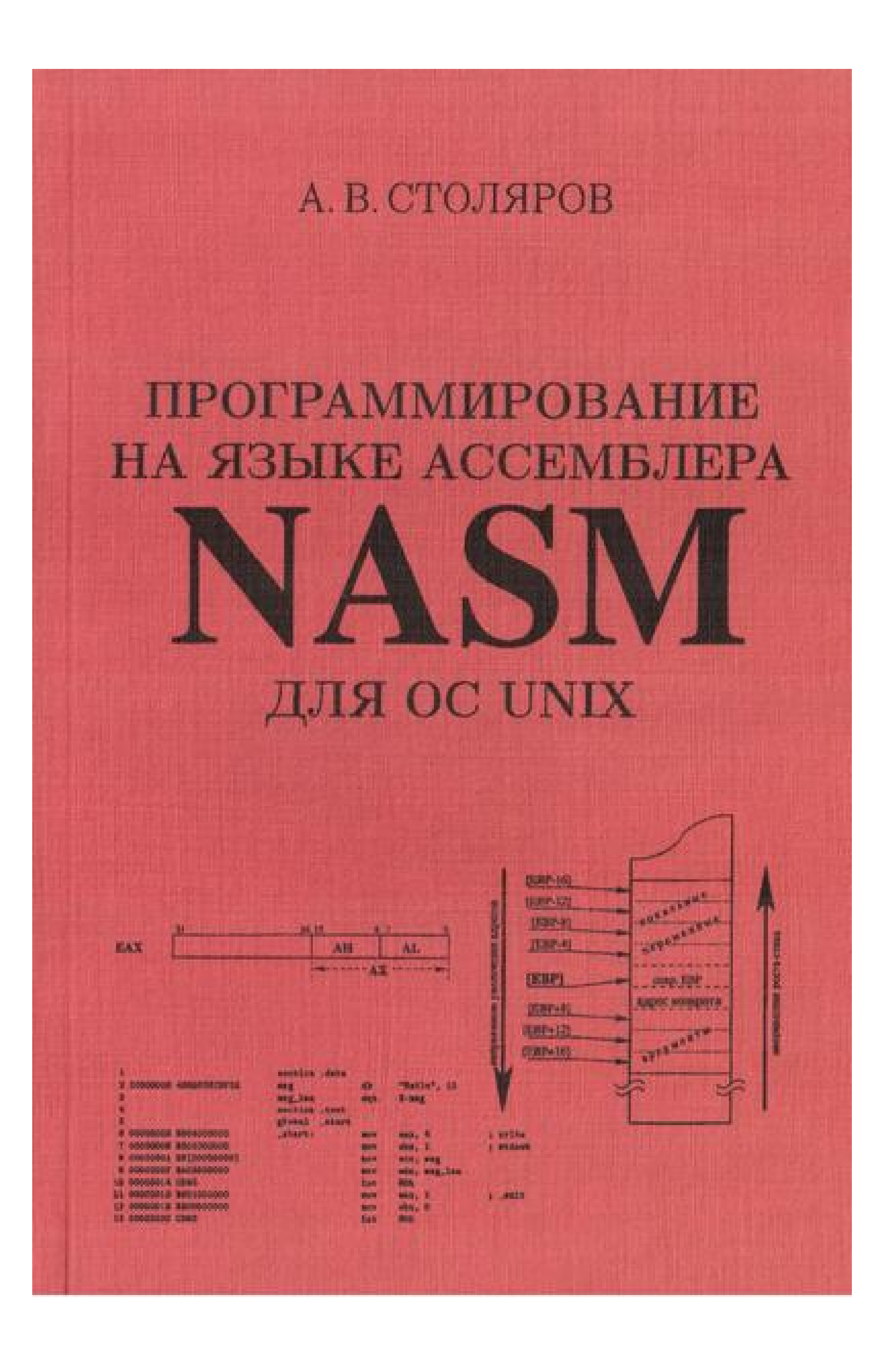

А. В. СТОЛЯРОВ
ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ АССЕМБЛЕРА
NASM
ДЛЯ ОС UNIX
EAX
AH
AL
AX

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

А. В. Столяров

Программирование на языке ассемблера *NASM* для ОС Unix

учебное пособие

издание второе, исправленное и дополненное

Москва
2011

УДК 004.431.4
ББК 32.973.26-018.1
С81

Столяров А. В.
С81 **Программирование на языке ассемблера NASM для ОС Unix**: Уч. пособие. – 2-е изд. – М.: МАКС Пресс, 2011. – 188 с.: ил.

ISBN 978-5-317-03627-0

В пособии изложены основы низкоуровневого программирования (программирования на уровне машинных команд) на примере ассемблера NASM для платформы i386 под управлением операционных систем семейства Unix (примеры расчитаны на Linux и FreeBSD) в «плоской» модели адресации памяти.

В курсе рассмотрены основы архитектуры фон Неймана, принципы машинного представления целых чисел и чисел с плавающей точкой, частично изложена система команд процессора i386, рассмотрены основы работы под управлением мультизадачной операционной системы, прямое использование системных вызовов. Изложены основы синтаксиса языка ассемблера NASM, описывается макропроцессор этого ассемблера; обсуждается раздельная трансляция, понятие объектного кода, работа редактора связей.

Для студентов программистских специальностей, преподавателей и всех желающих освоить низкоуровневое программирование.

УДК 004.431.4
ББК 32.973.26-018.1

Учебное издание
СТОЛЯРОВ Андрей Викторович
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА
NASM ДЛЯ ОС UNIX

Напечатано с готового оригинал-макета

Подписано в печать 29.03.2011 г.
Формат 60x90 1/16. Усл.печ.л. 11,5. Тираж 100 экз. Заказ 132.

Издательство ООО "МАКС Пресс"
Лицензия ИД № 00510 от 01.12.99 г.

11992 ГСП-2, Москва, Ленинские горы,
МГУ им. М.В.Ломоносова, 2-й учебный корпус, 627 к.
Тел. 939-3890, 939-3891. Тел./Факс 939-3891

ISBN 978-5-317-03627-0

Предисловие для преподавателей

В современной практике индустриального программирования языки ассемблера применяются сравнительно редко; для разработки низкоуровневых программ практически в любых ситуациях подходит язык Си, позволяющий достигать тех же целей многократно меньшими затратами труда, притом с аналогичной, а во многих случаях и более высокой эффективностью получаемого исполняемого кода (последнее достигается за счёт применения оптимизаторов). С помощью языков ассемблера сейчас реализуются разве что весьма специфические фрагменты ядер операционных систем и системных библиотек.

Тем не менее, изучение программирования на языке ассемблера является обязательным для студентов всех специальностей, связанных с программированием. Это легко объяснить: программист, не имеющий опыта работы на уровне команд процессора, попросту не ведает, что *на самом деле* творит. Вставляя в программу на языке высокого уровня те или иные операции, такой программист часто не догадывается, сколь сложную и ресурсоёмкую задачу он ставит перед процессором. На выходе мы имеем огромные программы, обескураживающие своей низкой эффективностью — например, приложения для автоматизации офисного документооборота, которым оказывается «тесно» в четырёх гигабайтах оперативной памяти и для которых оказывается «слишком медленным» процессор, на много порядков превосходящий по быстродействию суперкомпьютеры восьмидесятых годов.

Кроме того, иногда ссылки на реализацию на уровне машинных команд помогают объяснить студентам средства языков высокого уровня и библиотек. Так, приведя в качестве примера соответствующую ассемблерную вставку, можно наглядно показать различие между системным вызовом и его обёрткой в виде библиотечной функции. Хорошо помогают низкоуровневые иллюстрации также и для объяснения ситуаций состязания при работе с разделяемыми переменными, ими можно воспользоваться для рассказа о функциях `setjmp` и `longjmp` и для описания абстракций более высокого уровня, таких как виртуальные функции в объектно-ориентированном программировании или обработка исключений в языках, подобных Си++ или Аде. Студенту, имеющему представление о механизме стековых фреймов, оказывается гораздо проще объяснить совсем уж, казалось бы, далёкую от фоннеймановского вычислителя материю — оптимизацию остаточной рекурсии в Лиспе и других функциональных языках.

Таким образом, обучение программированию на языке ассемблера имеет своей целью не создание технических навыков собственно разработки с использованием ассемблеров, а скорее выработку понимания того, что же на самом деле представляет собой компьютер и как с его помощью следует решать задачи. С этой точки зрения не играет существенной роли выбор конкретной платформы, среды и собственно ассемблера: общие принципы работы центральных процессоров различаются мало. С другой стороны, понятно, что использовать следует настоящий компьютер, а не эмулятор, даже если это эмулятор реально существующего компьютера, поскольку программирование эмулятора оставляет у студентов ощущение «игрушечности» используемой среды, во многих слу-

чаях превращающееся в уверенность, что «с настоящим компьютером ничего бы не вышло».

Большинство существующих учебных пособий по программированию на языке ассемблера ориентировано на ранние процессоры серии 8086, так называемый «реальный» 16-битный режим работы, операционную среду MS DOS и один из хорошо известных с тех времён ассемблеров `tasm` или `masm`. Причины такого выбора хорошо понятны. С одной стороны, с появлением компьютеров линии IBM PC составителям и преподавателям соответствующих дисциплин в ВУЗах волей-неволей пришлось перейти именно на эту платформу, поскольку другие оказались недоступны; компьютеры, основанные на архитектуре 80x86, до сих пор остаются наиболее доступными для использования в компьютерных классах, практически исключая другие аппаратные платформы из рассмотрения. С другой стороны, с развитием линейки 80x86 возможность запуска программ в режиме эмуляции DOS сохранялась, что позволило сэкономить силы, не изучая архитектуру более новых процессоров линейки и не адаптируя под них существующие учебные курсы.

Между тем, в современных реалиях такой выбор платформы для изучения уже невозможно считать удачным. В самом деле, MS DOS как среда выполнения программ безнадёжно устарела ещё к середине 1990х годов; сам «реальный режим» на современных процессорах поддерживается на уровне микрокода, то есть фактически в режиме программной эмуляции, пусть и внутри процессора. Кроме того, с переходом к 32-битным процессорам (т. е. начиная с процессора 80386) система команд стала существенно более логичной, что подчёркивает бессмысленность траты учебного времени на объяснение странностей[1] архитектуры «реального режима» — странностей, которые заведомо никогда больше не появятся ни в одном процессоре.

Если говорить об использовании 32-битной системы команд (т. н. платформы i386), то выбор операционной среды оказывается сравнительно невелик, хотя и более разнообразен, нежели во времена MS DOS: это либо операционные системы линии MS Windows, либо представители семейства Unix. И здесь необходимо отметить, что при обучении основам программирования (причём это относится не только к программированию на языке ассемблера) крайне желательно наличие культуры консольных приложений. Написание консольных программ для операционной среды, в которой соответствующая культура отсутствует, создаёт уже знакомое ощущение «игрушечности» происходящего и, к сожалению, существенно расхолаживает студентов; начинать же обучение программированию с рисования окошек, что пришлось бы сделать для написания полноценных программ под Windows, категорически неприемлемо.

Кроме того, простой и прозрачный набор системных вызовов ОС Unix, логичные правила взаимодействия операционной системы с пользовательским процессом, использование в процессах «плоской» (flat) модели адресации памяти делают именно операционные системы семейства Unix, в особенности сво-

[1] В качестве примера можно назвать, прежде всего, систему адресов, состоящих из «сегмента» и «смещения», которая формирует несколько странное понимание термина «сегмент»; кроме того, следует упомянуть список допустимых регистровых пар в исполнительном адресе, ограничение команд условного перехода «короткими» прыжками и т. д.

бодно распространяемые (такие, как Linux и FreeBSD) заведомо более подходящими для ознакомления студентов со спецификой программирования на языке ассемблера.

Отдельно необходимо пояснить выбор конкретного ассемблера. Как известно, для работы с процессорами семейства x86 используются два основных подхода к синтаксису языка ассемблера — это синтаксис AT&T и синтаксис Intel. Одна и та же команда процессора представляется в этих синтаксических системах совершенно по-разному: например, команда, в синтаксисе Intel выглядящая как

```
mov eax, [a+edx]
```

в синтаксисе AT&T будет записываться следующим образом:

```
movl a(%edx), %eax
```

В среде ОС Unix традиционно более популярен именно синтаксис AT&T, но в применении к поставленной учебной задаче это создаёт некоторые проблемы. Учебные пособия, ориентированные на программирование на языке ассемблера в синтаксисе Intel, всё-таки существуют, тогда как синтаксис AT&T описывается исключительно в специальной (справочной) технической литературе, не имеющей целью обучение. Кроме того, необходимо учитывать и многолетнее господство среды MS DOS в качестве платформы для аналогичных учебных курсов; всё это позволяет назвать синтаксис Intel существенно более привычным для преподавателей (да и для некотрых студентов, как ни странно, тоже) и лучше поддерживаемым. В среде ОС Unix доступно два основных ассемблера, поддерживающих синтаксис Intel: это NASM («Netwide Assembler»), разработанный Саймоном Тетхемом и Джулианом Холлом, и FASM («Flat Assembler»), созданный Томашем Гриштаром. Сделать однозначный выбор между этими двумя ассемблерами оказывается достаточно сложно. В настоящем пособии рассматривается язык ассемблера NASM, в том числе и специфические для него макросредства; такой выбор не обусловлен никакими серьёзными причинами и попросту случаен.

Предисловие для студентов

Прежде чем приступать к изучению очередной дисциплины, желательно понять, зачем (с какой целью) эта дисциплина вообще изучается. В особенности это касается технических предметов, к которым, безусловно, относится и курс «Архитектура ЭВМ», в рамках которого обычно изучается программирование на языке ассемблера. Учебное пособие, которое вы держите в руках, ориентировано на программирование на языке ассемблера NASM в среде ОС Unix. Между тем, подавляющее большинство профессиональных программистов, услышав о таком, лишь усмехнётся и задаст риторический вопрос: «да кто же пишет под Unix на ассемблере? На дворе ведь XXI век!» Самое интересное, что при этом они будут совершенно правы. Особенно очевидной становится их правота, если вспомнить, что именно ОС Unix — первая в мире операционная

система, которая была написана на языке программирования высокого уровня, специально для этого придуманном (на языке Си). До появления ОС Unix считалось, что операционные системы можно писать только на языке ассемблера. Более того, в современном мире программирование на языке ассемблера оказалось вытеснено даже из такой традиционно «ассемблерной» области, как программирование микроконтроллеров — маленьких однокристалльных ЭВМ, предназначенных для встраивания во всевозможную технику, от стиральных машин и сотовых телефонов до самолётов и турбин на электростанциях. В большинстве случаев прошивки микроконтроллеров сейчас пишут тоже на Си, и лишь небольшие вставки выполняют на языке ассемблера.

Конечно, совсем обойтись без фрагментов на языке ассемблера пока не получается. Отдельные ассемблерные модули, а равно и ассемблерные вставки в текст на других языках присутствуют и в ядрах операционных систем, и в системных библиотеках того же языка Си (и других языков высокого уровня); в особых случаях программисты микроконтроллеров тоже вынуждены отказываться от Си и писать «на ассеблере», чтобы, например, сэкономить дефицитную память[2]. Однако такие случаи редки, и мало кому из вас, изучающих ныне программирование на языке ассемблера, придётся хотя бы один раз за всю жизнь прибегнуть к языку ассемблера на практике.

Так зачем же тратить время на изучение ассемблера? Ведь всё равно это никогда не пригодится? Так это выглядит лишь на первый взгляд; при более внимательном рассмотрении вопроса умение мыслить в терминах машинных команд не просто «пригодится», оно оказывается жизненно необходимым любому профессиональному программисту, даже если этот программист никогда не пишет на языке ассемблера. На каком бы языке вы ни писали свои программы, необходимо хотя бы примерно представлять, что конкретно будет делать процессор, чтобы исполнить вашу высочайшую волю. Если такого представления нет, программист начинает бездумно применять все доступные операции, не ведая, что *на самом деле* творит. Между тем, одно присваивание, записанное, скажем, на языке Си++, может выполниться в одну машинную команду, а может повлечь *миллионы* таких команд[3]. Два таких присваивания записываются в программе совершенно одинаково (знаком равенства), но этот факт никак нам не поможет.

Вообще, профессиональный пользователь компьютеров, будь то программист или системный администратор, может себе позволить что-то *не знать*, но ни в коем случае не может позволить себе *не понимать*, как устроена вычислительная система на всех её уровнях, от электронных логических схем до громоздких прикладных программ. Не понимая чего-то, мы оставляем в своём тылу место для «ощущения магии»: на каком-то глубоком, почти подсознательном уровне нам продолжает казаться, что что-то там не чисто и без парочки чародеев с волшебными палочками не обошлось. Такое ощущение для профессионала недопустимо категорически: напротив, профессионал обязан быть уверен, вплоть до глубоких слоёв подсознания, что то устройство, с которым он

[2]Например, некоторые микроконтроллеры имеют всего 256 байт оперативной памяти и 8 Кб псевдопостоянной памяти для хранения кода программы.

[3]Для знающих Си++ поясним: что будет, если применить операцию присваивания к объекту типа `list<string>`, содержащему пару тысяч элементов?

имеет дело, создано такими же людьми, как и он сам, и ничего «волшебного» или «непознаваемого» собой не представляет.

В этом плане совершенно не важно, какую конкретную архитектуру и язык какого конкретного ассемблера изучать. Зная один язык ассемблера, вы сможете начать писать на любом другом, потратив два-три часа (а то и меньше) на изучение справочной информации; но главное тут в том, что, умея мыслить в терминах машинных команд, вы всегда будете знать, что делаете, и всегда сможете понять, что происходит.

В заключение скажем пару слов о причинах выбора конкретной платформы. Машины на основе процессоров семейства i386 мы избрали исключительно из-за их широкого распространения. Что касается среды ОС Unix, то среди всех возможных операционных сред, имеющихся на платформе i386, именно программирование в ОС Unix оказывается *самым простым*, ну а лишние сложности нам ни к чему.

Итак, теперь вы знаете, что ответить скептикам по поводу программирования на языке ассемблера под ОС Unix. Правильным ответом будет фраза **«нам нужно было попрактиковаться в ассемблерном программировании *под какую-нибудь существующую систему*, всё равно какую, а ОС Unix мы выбрали, потому что под ней это делать проще всего»**. Отметим, что эта фраза будет нам полезна, даже если ни одного скептически настроенного профессионального программиста мы не встретим: действительно, ведь здесь одной фразой выражена и наша цель, и принципы, по которым мы выбирали средства.

Благодарности и посвящение

Автор глубоко признателен Владимиру Николаевичу Пильщикову, прочитавшему рукопись и сделавшему ряд ценнейших замечаний, которые позволили второму изданию этого пособия стать существенно лучше первого. Также автор считает своим приятным долгом поблагодарить Павла Сутырина, принявшего активное участие в вычитывании рукописи первого издания.

Первое издание этого пособия, предпринятое в 2010 году в МГТУГА, стало возможным исключительно благодаря заинтересованности, помощи и поддержке со стороны заведующего кафедрой прикладной математики МГТУГА профессора Валерия Леонидовича Кузнецова, которому автор хотел бы сказать огромное спасибо (и не только за это).

Излагаемый в пособии учебный материал был впервые апробирован автором в ходе чтения лекционного курса «Архитектура ЭВМ и язык ассемблера» первокурсникам Ташкентского филиала МГУ им. М. В. Ломоносова весной 2007 года. Самым талантливым и ярким из них — Линаре Адыловой, Максиму Болонкину, Юле Бутковой, Алисе Киреевой и другим — автор с величайшим удовольствием посвящает это пособие.

Глава 1. Введение

§ 1.1. Машинный код и ассемблер

Практически все современные цифровые вычислительные машины работают по одному и тому же принципу. Вычислительное устройство (собственно сам компьютер) состоит из ***центрального процессора***, ***оперативной памяти*** и ***периферийных устройств***. В большинстве случаев все эти компоненты подключаются к ***общей шине*** — устройству из множества параллельных проводов (дорожек на печатной плате), позволяющему компонентам компьютера обмениваться информацией между собой.

Оперативная память состоит из одинаковых ***ячеек памяти***, каждая из которых имеет свой уникальный номер, называемый ***адресом***. Ячейка содержит несколько (чаще всего — восемь) двоичных разрядов, каждый из которых может находиться в одном из двух состояний, обычно обозначаемых как «ноль» и «единица». Это позволяет ячейке как единому целому находиться в одном из 2^n состояний, где n — количество разрядов в ячейке; так, если разрядов восемь, то возможных состояний ячейки будет $2^8 = 256$, или, иначе говоря, ячейка может «помнить» число от 0 до 255. Если требуется хранить число из бóльшего диапазона, используют несколько идущих подряд ячеек памяти. Отметим, что при рассмотрении нескольких соседних ячеек как представления одного целого числа на разных машинах используют два разных подхода к порядку следования байтов. Один подход, называемый *little-endian*[1], предполагает, что первым идёт самый младший байт числа, далее в порядке возрастания, и самый старший байт идёт последним. Второй подход, который называют *big-endian*, прямо противоположен: сначала идёт старший байт числа, а

[1] «Термины» big-endians и little-endians введены Джонатаном Свифтом в книге «Путешествия Гулливера» для обозначения непримиримых сторонников разбивания яиц соответственно с тупого конца и с острого. На русский язык эти названия обычно переводились как *тупоконечники и остроконечники*. Аргументы в пользу той или иной архитектуры действительно часто напоминают священную войну остроконечников с тупоконечниками.

младший располагается в памяти последним. Процессоры, которые мы будем рассматривать, относятся к категории «little-endian», то есть хранят младший байт первым.

При необходимости содержимое ячейки памяти можно рассматривать и как строчку из отдельных двоичных разрядов (битовую строку), и другими способами: например, достаточно сложный способ интерпретации значений двоичных разрядов используется для хранения дробных чисел, так называемых ***чисел с плавающей точкой***. Кроме того, содержимое ячейки памяти (или нескольких ячеек, идущих подряд) может быть истолковано как ***машинная инструкция*** — кодовое число, идентифицирующее одну из множества операций, которые может выполнять центральный процессор.

Важно понимать, что сама по себе ячейка памяти «не знает», как именно следует интерпретировать хранящуюся в ней информацию. Рассмотрим это на простейшем примере. Пусть у нас есть четыре идущие подряд ячейки памяти, содержимое которых соответствует шестнадцатеричным числам `41`, `4E`, `4E` и `41` (соответствующие десятичные числа — 65, 79, 79, 65). Информацию, содержащуюся в такой области памяти, можно с совершенно одинаковым успехом истолковать:

- как целое число 1095650881;
- как дробное число (т. н. число с плавающей точкой) 12.894105;
- как текстовую строку, содержащую имя `'ANNA'`;
- и, наконец, как последовательность машинных команд; в частности, на процессорах платформы i386 это будут команды, условно обозначаемые `inc ecx`, `dec esi`, `dec esi`, `inc ecx`. Что делают эти команды, мы узнаем позже.

В процессоре имеется некоторое количество ***регистров*** — схем, напоминающих ячейки памяти; поскольку регистры находятся непосредственно в процессоре, они работают очень быстро, но их количество ограничено, так что использовать регистры следует для хранения самой необходимой информации. Процессор обладает способностью копировать данные из оперативной памяти в регистры и обратно, производить над содержимым регистров арифметические и другие операции; в некоторых случаях операции можно производить и непосредственно с данными в ячейках памяти, не копируя их содержимое в регистры[2].

---

[2]Наличие или отсутствие такой возможности зависит от конкретного процессора; так, процессоры Pentium могут, минуя регистры, прибавить заданное число к содержимому заданной ячейки памяти и произвести некоторые другие операции, тогда как процессоры SPARC, применявшиеся в компьютерах фирмы Sun Microsystems, могут только копировать содержимое ячейки памяти в регистр или, наоборот, содержимое

Количество информации, которую может обработать процессор в один приём (за одну команду), называется ***машинным словом***. Размер большинства регистров в точности равен машинному слову. В современных системах машинное слово, как правило, больше, чем ячейка памяти; так, машинное слово процессора Pentium составляет 32 бита, то есть четыре восьмибитовые ячейки памяти.

Здесь необходимо сделать одно важное замечание. Процессор Pentium является очередным представителем *линейки процессоров* x86, и ранние представители этой линейки (вплоть до процессора 80286) были 16-разрядными, то есть их машинное слово составляло 16 бит. Программисты, работающие с этими процессорами на уровне языка ассемблера, привыкли называть «словом» именно два байта информации, а четыре байта называли «двойным словом», и в языках ассемблера использовали соответствующие обозначения (`word` и `dword`). Когда с выходом очередного процессора размер слова удвоился, программисты не стали менять привычную терминологию, что порождает определённую путаницу. К этому вопросу мы ещё вернёмся.

Программа, предназначенная к выполнению, записывается в оперативную память в виде последовательности машинных инструкций (команд), то есть цифровых кодов, обозначающих те или иные операции. Один из регистров процессора, так называемый ***счётчик команд***[3], содержит адрес той ячейки памяти, в которой располагается следующая инструкция, предназначенная к выполнению.

Процессор работает, раз за разом выполняя ***цикл обработки команды***. В начале этого цикла из ячеек памяти, на которые указывает[4] счётчик команд, считывается код очередной команды. Сразу после этого счётчик команд меняет своё значение так, чтобы указывать на следующую команду в памяти; например, если только что прочитанная команда занимала три ячейки памяти, то счётчик команд увеличивается на три. Схемы процессора дешифруют код и выполняют действия, предписанные этим кодом: например, это может быть предписание «скопировать число из одного регистра в другой» или «взять содержимое регистра A, прибавить к нему содержимое регистра B, а результат поместить обратно в регистр A», и т. п. Когда действия, предписанные командой, будут исполнены, процессор вновь возвращается к началу цикла обработки команд, так что следующий проход этого цикла выполняет уже следующую команду, и так далее до бесконечности (точнее, пока процессор не выключат).

регистра в ячейку памяти, но никаких других операций над ячейками памяти выполнять не могут.

[3] Английское название этого регистра — *instruction pointer*, то есть «указатель на инструкцию»; устоявшийся в русскоязычной литературе термин «счётчик команд» не столь удачен, ведь этот «счётчик» на самом деле ничего не считает.

[4] Выражение вида «нечто указывает на ячейку памяти» является синонимом выражения «нечто содержит адрес ячейки памяти».

Некоторые машинные команды могут изменить последовательность выполнения команд, предписав процессору перейти в другое место программы (то есть, попросту говоря, в явном виде изменить текущее значение счётчика команд). Такие команды называются ***командами перехода***. Различают ***условные*** и ***безусловные*** переходы; команда условного перехода сначала проверяет истинность некоторого условия и производит переход, только если условие выполнено, тогда как команда безусловного перехода просто заставляет процессор продолжить выполнение команд с заданного адреса без всяких проверок. Процессоры обычно поддерживают также переходы с запоминанием точки возврата, которые используются для вызова подпрограмм.

Ясно, что программа, которую выполняет компьютер, должна быть представлена в виде, понятном центральному процессору; такое представление называется ***машинным кодом***. Программа в машинном коде состоит из отдельных ***машинных команд***, которые обозначаются числами (кодами). Процессор легко может дешифровать такие коды команд, но человеку их запомнить очень трудно, тем более что во многих случаях нужное число приходится *вычислять*, подставляя в определённые места кодовые цепочки двоичных битов. Вот, например, два байта, записываемые в шестнадцатеричной системе как `01 D8` (соответствующие десятичные значения — 1, 216) обозначают на процессорах Pentium команду «взять число из регистра `EAX`, прибавить к нему число из регистра `EBX`, результат сложения поместить обратно в регистр `EAX`». Запомнить два числа `01 D8` несложно, но ведь разных команд на процессоре Pentium — несколько сотен, да к тому же здесь сама команда — только первый байт (`01`), а второй (`D8`) нам придётся вычислить в уме, вспомнив (или узнав из справочника), что младшие три бита в этом байте обозначают первый регистр (первое слагаемое, а также и место, куда следует записать результат), следующие три бита обозначают второй регистр, а самые старшие два бита здесь должны быть равны единицам, что означает, что оба операнда являются регистрами. Зная (или, опять же, подсмотрев в справочнике), что номер регистра `EAX` — 0, а номер регистра `EBX` — 3, мы теперь можем записать двоичное представление нашего байта: `11 011 000` (пробелы вставлены для наглядности), что и даёт в десятичной записи 216, а в шестнадцатеричной — искомое `D8`.

Если нам потребуется освежить в памяти кусочек нашей программы, написанный два дня назад, то чтобы его прочитать, нам придётся вручную раскладывать байты на составляющие их биты и, сверяясь со справочником, вспоминать, что же какая команда делает. Очевидно, что, если программиста заставить составлять программы вот таким вот способом, ничего полезного он не напишет за всю свою жизнь, тем более что в любой, даже самой небольшой, но практически применимой программе таких команд будет несколько тысяч, ну а самые большие программы

состоят из *сотен миллионов* машинных команд.

При работе с языками программирования высокого уровня, такими как Паскаль, Си, Лисп и др., программисту предоставляется возможность написать программу в виде, понятном и удобном для человека, а не для центрального процессора. В этом случае приходится применять ***компилятор*** — программу, принимающую на вход текст программы на языке программирования высокого уровня и выдающую эквивалентный машинный код[5]. Программирование на языках высокого уровня удобно, но, к сожалению, не всегда применимо. Причины этого могут быть самые разные. Например, язык высокого уровня может не учитывать некоторые особенности конкретного процессора, либо программиста может не устраивать тот конкретный способ, которым компилятор реализует те или иные конструкции исходного языка с помощью машинных кодов. В этих случаях приходится отказаться от языка высокого уровня и составить программу в виде *конкретной последовательности машинных команд*. Однако, как мы уже видели, составлять программу непосредственно в машинных кодах очень и очень сложно. И здесь на помощь приходит программа, называемая ***ассемблером***.

Ассемблер — это программа, принимающая на вход текст, содержащий условные обозначения машинных команд, удобные для человека, и переводящий эти обозначения в последовательность соответствующих кодов машинных команд, понятных процессору. В отличие от самих машинных команд, их условные обозначения, называемые также ***мнемониками***, запомнить сравнительно легко. Так, команда из вышеприведённого примера, код которой, как мы с некоторым трудом выяснили, равен `01 D8`, в условных обозначениях[6] выглядит так:

```
add  eax, ebx
```

Здесь нам уже не надо заучивать числовой код команды и вычислять в уме обозначения операндов, достаточно запомнить, что словом `add` обозначается сложение, причём в таких случаях всегда первым после обозначения команды стоит первое слагаемое (не обязательно регистр, это может быть и область памяти), вторым — второе слагаемое (это может быть и регистр, и область памяти, и просто число), а результат всегда заносится на место первого слагаемого. Язык таких условных обозначений (мнемоник) называется ***языком ассемблера***.

Программирование на языке ассемблера коренным образом отличается от программирования на языках высокого уровня. На языке высокого

[5]Вообще говоря, компилятор — это программа, переводящая программы с одного языка на другой; перевод на язык машинного кода — это лишь частный случай, хотя и очень важный.

[6]Здесь и далее используются условные обозначения, соответствующие ассемблеру NASM, если не сказано иное.

уровня (на том же Паскале) мы задаём лишь общие указания, а компилятор волен сам выбирать, каким именно способом их исполнить — например, какими регистрами и ячейками памяти воспользоваться для хранения промежуточных результатов, какой применить алгоритм для выполнения какой-нибудь нетривиальной инструкции, и т. д. С целью оптимизации быстродействия компилятор может переставить инструкции местами, заменить одни на другие — лишь бы результат остался неизменным. В отличие от этого, **на языке ассмеблера мы совершенно однозначно и недвусмысленно указываем, из каких машинных команд будет состоять наша программа, и никакой свободы ассемблер (в отличие от компилятора языка высокого уровня) не имеет**.

В отличие от машинных кодов, мнемоники доступны для человека, то есть программист может работать с мнемониками без особого труда, но это не означает, что программировать на языке ассемблера просто. Действие, на описание которого мы бы потратили один оператор языка высокого уровня, может потребовать десятка, если не сотни строк на языке ассемблера, а в некоторых случаях и больше. Дело тут в том, что компилятор языка высокого уровня содержит большой набор готовых «рецептов», как решать часто возникающие небольшие задачи, и предоставляет все эти «рецепты» программисту в виде удобных высокоуровневых конструкций; ассемблер же никаких таких рецептов не содержит, так что в нашем распоряжении оказываются только возможности процессора.

Интересно, что для одного и того же процессора может существовать несколько разных ассемблеров. На первый взгляд это кажется странным, ведь не может же один и тот же процессор работать с разными системами машинных кодов (так называемыми ***системами команд***). В действительности ничего странного здесь нет, достаточно вспомнить, что же такое на самом деле ассемблер. Система команд процессора, разумеется, не может измениться (если только не взять другой процессор). Однако для одних и тех же команд можно придумать разные обозначения; так, уже знакомая команда `add eax,ebx` в обозначениях, принятых в компании AT&T, будет выглядеть как `addl %ebx,%eax` — и мнемоника другая, и регистры не так обозначены, и операнды не в том порядке, хотя получаемый машинный код, разумеется, строго тот же самый — `01 D8`. Кроме того, при программировании на языке ассемблера мы обычно пишем не только мнемоники машинных команд, но и ***директивы***, представляющие собой прямые приказы ассемблеру. Следуя таким указаниям, ассемблер может зарезервировать память, объявить ту или иную метку видимой из других модулей программы, перейти к генерации другой секции программы, вычислить (прямо во время ассемблирования) какое-нибудь выражение и даже сам (следуя, разумеется, нашим указаниям) «напи-

сать» фрагмент программы на языке ассемблера, который сам же потом и обработает. Набор таких вот директив, поддерживаемых ассемблером, тоже может быть разным, как по возможностям, так и по синтаксису.

Поскольку ассемблер — это не более чем программа, написанная вполне обыкновенными программистами, никто не мешает другим программистам написать свою программу-ассемблер, что часто и происходит. Ассемблер NASM, упоминаемый в названии данного пособия — это один из ассемблеров, существующих для процессоров семейства 80x86. Существуют и другие ассемблеры; возможно даже, что какой-нибудь из них покажется вам более удобным. На самом деле, не так уж важно, язык какого конкретного ассемблера изучать. Важно понять общий принцип работы на уровне команд процессора, и после этого вы сможете без труда освоить не только другой ассемблер, но и любой другой процессор с совсем другими командами.

§ 1.2. Особенности программирования под управлением мультизадачных операционных систем

Поскольку мы собираемся запускать написанные нами программы под управлением ОС Unix, нелишним будет заранее описать некоторые особенности таких систем с точки зрения выполняемых программ; эти особенности распространяются на все программы, выполняющиеся как под операционными системами семейства Unix, так и под многими другими системами, и никак не зависят от используемого языка программирования, но при работе на языке ассемблера становятся особенно заметны.

Практически все современные операционные системы позволяют запускать несколько программ на одновременное исполнение. Такой режим работы вычислительной системы, называемый ***мультизадачным***[7], порождает некоторые проблемы, требующие решения со стороны аппаратуры (прежде всего — центрального процессора).

Во-первых, необходимо защитить выполняемые программы друг от друга и саму операционную систему от пользовательских программ. Если (пусть даже не по злому умыслу, а по ошибке) одна из выполняемых задач изменит что-то в памяти, принадлежащей другой задаче, скорее всего это приведёт к аварии этой второй задачи, причём найти причину такой аварии окажется принципиально невозможно. Если пользовательская задача (опять-таки по ошибке) внесёт изменения в память опе-

[7]Термин «задача», строго говоря, довольно сложен, но упрощённо задачу можно понимать как «программу, которая запущена на выполнение под управлением операционной системы», иначе говоря, при запуске программы в системе возникает задача.

рационной системы, это приведёт уже к аварии всей системы, причём, опять-таки, без малейшей возможности разобраться в причинах таковой. Поэтому центральный процессор должен поддерживать механизм ***защиты памяти***: каждой выполняющейся задаче выделяется определённая область памяти, и к ячейкам за пределами такой области задача обращаться не может.

Во-вторых, в мультизадачном режиме пользовательские задачи, как правило, не допускаются к прямой работе с внешними устройствами[8]. Если бы это правило не выполнялось, задачи постоянно начинали бы конфликтовать за доступ к устройствам, и такие конфликты, разумеется, приводили бы к авариям. Чтобы ограничить возможности пользовательской задачи, создатели центрального процессора объявили часть имеющихся машинных инструкций ***привилегированными***. Процессор может работать либо в ***привилегированном режиме***, также называемом ***режимом суперпользователя***, либо в ***ограниченном режиме***, который также называют ***режимом задачи*** или ***пользовательским режимом***[9]. В ограниченном режиме привилегированные команды недоступны. В привилегированном режиме процессор может выполнять все имеющиеся инструкции, как обычные, так и привилегированные. Операционная система выполняется, естественно, в привилегированном режиме, а при передаче управления пользовательской задаче переключает режим в ограниченный. Вернуться в привилегированный режим процессор может только при условии одновременной передачи управления назад операционной системе, так что код пользовательской программы выполняться в привилегированном режиме не может. К категории привилегированных относятся инструкции, осуществляющие взаимодействие с внешними устройствами; также в эту категорию попадают инструкции, используемые для настройки механизмов защиты памяти и некоторые другие команды, влияющие на работу всей системы в целом. Все такие «глобальные» действия являются прерогативой операционной системы. **При работе под управлением мультизадачной операционной системы пользовательская задача может лишь преобразовывать информацию в отведённой ей области оперативной памяти. Всё взаимодействие с внешним миром задача производит через обращения к операционной системе.** Даже просто вывести на экран строку задача самостоятельно не может, ей необходимо попросить об этом операционную систему. Такое обращение пользова-

[8]Из этого правила есть исключения, связанные, например, с отображением графической информации на дисплее, но в этом случае устройство должно быть закреплено за одной пользовательской задачей и строго недоступно для других задач.

[9]На самом деле процессор i386 и его потомки имеют не два, а четыре режима, называемые также *кольцами защиты*, но реально операционные системы используют только нулевое кольцо (высший возможный уровень привилегий) и третье кольцо (низший уровень привилегий).

тельской задачи к операционной системе за теми или иными услугами называется ***системным вызовом***. Интересно, что *завершение задачи* тоже может выполнить только операционная система. Таким образом, корректная пользовательская задача обойтись без системных вызовов не может никак, ведь даже просто завершиться она может только с помощью соответствующего системного вызова.

Ещё один важный момент, который необходимо упомянуть перед началом изучения конкретного процессора — это наличие в нашей операционной среде механизма ***виртуальной памяти***. Попробуем понять, что это такое. Как уже говорилось, оперативная память делится на одинаковые по своей ёмкости *ячейки* (в нашем случае каждая ячейка содержит 1 байт данных), и каждая такая ячейка имеет свой порядковый номер. Именно этот номер использует центральный процессор для работы с ячейками памяти через общую шину, чтобы отличать их одну от другой. Назовём этот номер ***физическим адресом*** ячейки памяти. Изначально никаких других адресов, кроме физических, у ячеек памяти не было. В машинном коде программ использовались именно физические адреса, которые называли просто «адресами», без уточняющего слова «физический». Однако с развитием мультизадачного режима работы вычислительных систем оказалось, что в силу целого ряда причин использование физических адресов *неудобно*. Например, программа в машинном коде, в которой используются физические адреса ячеек памяти, не сможет работать в другой области памяти — а ведь в мультизадачной ситуации может оказаться, что нужная нам область уже занята другой задачей. Есть и другие причины, которые обычно подробно рассматриваются в учебных курсах, посвященных операционным системам.

В современных процессорах используется два вида адресов. Сам процессор работает с памятью, используя уже знакомые нам физические адреса. А вот в программах, которые на процессоре выполняются, используются совсем другие адреса — *виртуальные*. ***Виртуальный адрес*** — это число из некоторого абстрактного ***виртуального адресного пространства***. На тех процессорах, с которыми мы будем работать, виртуальные адреса представляют собой 32-битные целые числа, то есть виртуальное адресное пространство есть множество целых чисел от 0 до $2^{32} - 1$; адреса обычно записываются в шестнадцатеричной системе, так что адрес может быть числом от `00000000` до `ffffffff`. Важно понимать, что виртуальный адрес совершенно не обязан соответствовать какой-либо ячейке памяти. Точнее говоря, *некоторые* виртуальные адреса соответствуют физическим ячейкам памяти, другие — не соответствуют, а некоторые адреса и вовсе могут то соответствовать физической памяти, то не соответствовать. Такие соответствия задаются путём соответствующей настройки центрального процессора; за эту настройку отвечает операционная система. Будучи соответствующим образом

настроенным, центральный процессор, получив из очередной машинной инструкции виртуальный адрес, *преобразует* его в адрес физический, и тогда уже обращается к оперативной памяти. Таким образом, мы в программах используем в качестве адресов не физические номера ячеек памяти, а некие абстрактные адреса, которые потом уже сам процессор преобразует в настоящие номера ячеек. Это позволяет, например, каждой программе иметь своё собственное адресное пространство: действительно, никто не мешает операционной системе настроить преобразования адресов так, чтобы один и тот же виртуальный адрес в одной пользовательской задаче отображался на одну физическую ячейку, а в другой задаче — на совсем другую.

Вопросы, связанные с созданием новых операционных систем, мы в нашем пособии рассматривать не будем. Вместо этого мы ограничимся рассмотрением возможностей процессора i386, доступных пользовательской задаче, работающей в ограниченном режиме. Более того, даже эти возможности мы рассмотрим не все; дело в том, что операционные системы семейства Unix выполняют пользовательские задачи в так называемой ***плоской модели*** адресации памяти, в которой не используется часть регистров и некоторые виды машинных команд. На изучение этих регистров и команд мы не будем тратить время, поскольку всё равно не сможем их применить. Позже в нашем курсе мы подробно рассмотрим механизмы взаимодействия с операционной системой, включая и способы организации системного вызова для систем Linux и FreeBSD; однако пока нам эти механизмы не известны, мы будем осуществлять ввод/вывод и завершение программы с помощью готовых *макросов* — специальных идентификаторов, которые наш ассемблер развернёт в целые последовательности машинных команд и уже в таком виде оттранслирует. Отметим, что к концу нашего курса мы сами научимся при необходимости создавать такие макросы.

§ 1.3. Машинное представление целых чисел

Подавляющее большинство компьютеров, созданных за всю историю вычислительной техники, обрабатывало и обрабатывает любую информацию, представляя её в двоичной системе, то есть в виде последовательности разрядов, каждый из которых может содержать одно из двух возможных значений (включено/выключено), обозначаемых обычно цифрами `0` и `1`. Разумеется, это касается и целых чисел; читатель, несомненно[10], уже знаком с двоичной системой счисления, в которой для записи чисел используются только две цифры (всё те же ноль и единица). Впрочем,

[10]Отметим на всякий случай, что изучение двоичной системы и других позиционных систем счисления входит в обязательную программу для средних школ.



Рис. 1.1. Механический счётчик

компьютеры, будучи реально существующими техническими устройствами, накладывают некоторые ограничения на представление целых чисел. Из математики мы знаем, что ряд чисел бесконечен, то есть каково бы ни было число N, всегда существует *следующее* число $N+1$. Для этого и количество знаков в записи числа, какую бы систему мы ни использовали, не должно никак ограничиваться — но вот как раз это требование исполнить технически невозможно (даже чисто теоретически: ведь количество атомов во вселенной считается конечным).

§ 1.3.1. Беззнаковые числа

На практике для компьютерного представления целого числа выделяется некоторое фиксированное[11] количество разрядов (бит); обычно это 8 бит (одна ячейка), 16 бит (две ячейки), 32 бита (четыре ячейки) или 64 бита (восемь ячеек). Ограничение разрядности приводит к появлению «наибольшего числа», причём это касается не только двоичной системы. Представьте себе, например, простое счётное устройство, используемое в электрических счётчиках и механических спидометрах старых автомобилей: цепочку роликов, на которых нанесены цифры и которые могут прокручиваться, а проходя через «точку переноса» (с девятки на ноль), прокручивают на единицу следующий ролик. Допустим, такое устройство состоит из пяти роликов (см. рис. 1.1). Сначала мы видим на нём число ноль: `00000`. По мере прокручивания правого ролика число будет меняться, мы увидим `00001`, потом `00002`, и так далее вплоть до `00009`. Если теперь провернуть самый правый ролик ещё на единичку, мы снова увидим в правой позиции ноль, но при этом самый правый ролик зацепит своего соседа слева и заставит его провернуться на единичку, так что мы увидим `00010`, то есть число десять; мы наблюдали при этом хорошо

[11]Некоторые языки программирования высокого уровня позволяют оперировать сколь угодно большими целыми числами, лишь бы хватило памяти; но такие возможности всегда реализуются программно (и довольно сложно), а мы сейчас рассматриваем программирование на низком уровне, которое отталкивается от возможностей процессора.

известный с младших классов *перенос*: «девять плюс один, ноль пишем, один в уме». То же самое произойдёт при переходе от числа 00019 к числу 00020, и так далее, а когда мы увидим число 00099 и прокрутим правый ролик ещё на единичку, в зацепление попадут сразу два его соседа, так что на единицу вперёд прокрутятся сразу три ролика, и мы получим число сто: 00100.

Теперь уже несложно будет понять, откуда берётся такой монстр, как «наибольшее возможное число»: рано или поздно наш счётчик досчитает до 99999, и теперь увеличивать число окажется некуда; когда мы в очередной раз прокрутим правый ролик на единицу вперёд, он зацепит за собой все остальные ролики, так что они все перейдут на следующую цифру, и мы снова увидим одни нули. Если бы у нас слева был ещё один ролик, он бы зацепился и показал единичку, так что результат бы был 100000 (что совершенно правильно), но у нас всего пять роликов, шестого нет. Такая ситуация называется ***перенос в несуществующий разряд***. Ясно, что такая ситуация не может возникнуть, когда мы пишем числа на бумаге: всегда можно дописать ещё одну цифру слева; когда же число представлено состоянием некой группы технических устройств, будь то цепочка роликов или набор триггеров в оперативной памяти компьютера, возможности приделать к числу ещё одну цифру у нас нет.

При использовании двочиной системы счисления происходит примерно то же самое с той разницей, что используется всего две цифры. Допустим, мы используем для подсчёта каких-нибудь предметов или событий ячейку памяти, которая содержит восемь разрядов. Сначала в ячейке ноль: 00000000. Добавив единицу, получаем двоичное представление единицы: 00000001. Добавляем ещё одну единицу, младший (самый правый) разряд увеличивать некуда, поскольку у нас только две цифры, поэтому он снова становится нулевым, но при этом происходит перенос, в результате которого единица появляется во втором разряде: 00000010; это двоичное представление числа 2. Дальше будет 00000011, 00000100 и так далее; но в какой-то момент во всех имеющихся разрядах окажутся единицы, так что прибавлять дальше будет некуда: 11111111; это двоичное представление числа 255 ($2^8 - 1$). Если теперь прибавить ещё единицу, вместо числа 256 мы получим «все нули», то есть просто ноль; произошел уже знакомый нам перенос в несуществующий разряд. Вообще, **при использовании для представления целых положительных чисел позиционной несмешанной системы счисления по основанию N и ограничении количества разрядов числом k максимальное число, которое мы можем представить, составляет** $N^k - 1$; так, в нашем примере со счётчиком было пять разрядов десятичной системы, и максимальным числом оказалось $99999 = 10^5 - 1$, а в примере с восьмибитной ячейкой система использовалась двоичная, разрядов было восемь, так что максимальным числом оказалось $2^8 - 1 = 255$.

§ 1.3.2. Знаковые числа; дополнительный код

Посмотрим теперь, как быть, если нужны не только положительные числа. Ясно, что нужен какой-то другой способ интерпретации комбинаций двоичных разрядов, такой, чтобы какие-то из комбинаций считались представлением отрицательных чисел. Будем в таких случаях говорить, что ячейка или область памяти хранит *знаковое целое число*, в отличие от предыдущего случая, когда говорят о *беззнаковом целом числе*.

На заре вычислительной техники для представления отрицательных целых чисел пытались использовать разные подходы, например, хранить знак числа как отдельный разряд. Оказалось, однако, что при этом неудобно реализовывать даже самые простые операции — сложение и вычитание, потому что приходится учитывать знаковый бит обоих слагаемых. Поэтому создатели компьютеров достаточно быстро пришли к использованию так называемого ***дополнительного кода***[12]. Если отрицательные числа представлять этим способом, сложение и вычитание реализуется на аппаратном уровне абсолютно одинаково вне зависимости от знаков слагаемых и даже от самого факта их «знаковости»: мы можем по-прежнему рассматривать все возможные битовые комбинации как представление неотрицательных чисел (то есть вернуться к беззнаковой арифметике), и схематически сложение и вычитание от этого не изменятся. Больше того, отпадает вообще надобность в отдельном электронном устройстве для вычитания: операция вычитания может быть реализована как операция прибавления числа, которому сначала сменили знак, причём это, как ни парадоксально, работает и для беззнаковых чисел.

Чтобы понять, как устроен дополнительный код, вернёмся к нашему примеру с механическим счётчиком. В большинстве случаев такие роликовые цепочки умеют крутиться как вперёд, так и назад, и если прокрутка вперёд давала нам прибавление единицы, то прокрутка назад будет выполнять вычитание единицы. Пусть теперь у нас все ролики выставлены на ноль и мы откручиваем счётчик назад. Результатом этого будет 99999; оно и понятно, ведь когда мы к 99999 прибавили единицу, то получилось 00000, а теперь мы проделали обратную операцию. Говорят, что у нас произошел ***заём из несуществующего разряда***: как и в случае с переносом в несуществующий разряд, если бы у нас был ещё один ролик, всё бы было правильно (например, $100000 - 1 = 99999$), но его нет. То же самое происходит и в двоичной системе: если во всех разрядах ячейки были нули (`00000000`) и мы вычли единицу, получим все единицы: `11111111`; если теперь снова прибавить единицу, мы снова

[12]Английский термин — two's complement, то есть «двоичное дополнение»; надо сказать, что ничего нового в использовании этого метода не было, метод десятичных дополнений использовал ещё Блез Паскаль для выполнения вычитаний на своем арифмометре.

получим нули во всех разрядах. Это логично приводит нас к идее **использовать в качестве представления числа -1 единицы во всех разрядах двоичного числа**, сколько бы ни было у нас таких разрядов. Так, если мы работаем с восьмиразрядными числами, 11111111 у нас теперь означает -1, а не 255; если мы работаем с шестнадцатиразрядными числами, 1111111111111111 теперь будет обозначать, опять-таки, -1, а не 65535, и так далее.

Продолжая операцию по вычитанию единицы над восьмиразрядной ячейкой, мы придём к заключению, что для представления числа -2 нужно использовать 11111110 (раньше это было число 254), для представления -3 — 11111101 (раньше это было 253), и так далее. Иначе говоря, мы волюнтаристски объявили часть комбинаций двоичных разрядов представляющими отрицательные числа вместо положительных, причём всегда новое (отрицательное) значение комбинации разрядов получается из старого (положительного) путём вычитания из него числа 256: $255 - 256 = -1$, $254 - 256 = -2$ и т. д. (число 256 представляет собой 2^8, а наши рассуждения верны только для частного случая с восьмиразрядными числами; в общем случае из старого значения нужно вычитать число 2^n, где n — используемая разрядность). Остаётся вопрос, в какой момент остановиться, то есть перестать считать числа отрицательными; иначе, увлёкшись, мы можем дойти до 00000001 и заявить, что это вовсе не 1, а -255. Принято следующее соглашение: **если набор двоичных разрядов рассматривается как представление знакового числа, то *отрицательными* считаются комбинации, старший бит которых равен 1**, а остальные комбинации считаются положительными. Таким образом, наибольшее по модулю отрицательное число будет представлено одной единицей в старшем разряде и нулями во всех остальных; в восьмибитном случае это 10000000, -128. Если из этого числа вычесть единицу, получится 01111111; эта комбинация (старший ноль, остальные единицы) считается представлением *наибольшего знакового числа* и для восьмибитного случая представляет, как несложно видеть, число 127. Как вы уже догадались, прибавление единицы к этому числу снова даст наибольшее по модулю отрицательное. Переход через границу между комбинациями 011...11 и 100...00 для знаковой целочисленной арифметики[13] представляет собой аналог переноса и займа для несуществующего разряда, который мы наблюдали в арифметике беззнаковой, но называется эта ситуация иначе: ***переполнение***.

Именно такое, а не какое-либо другое расположение границы переполнения даёт две приятные возможности. Во-первых, знак числа можно определить, взяв от него всего один (старший) бит. Во-вторых, оказывается очень простой операция *смены знака числа*. **Чтобы сменить знак**

[13]То есть когда сумма двух положительных оказывается отрицательной и наоборот.

числа на противоположный при использовании дополнительного кода, достаточно сменить значения во всех разрядах на противоположные, а к полученному значению прибавить единицу. Например, число 5 представляется следующим восьмибитным знаковым целым: 00000101. Чтобы получить представление числа -5, мы сначала инвертируем все разряды, получаем 11111010; теперь прибавляем единицу и получаем 11111011, это и есть представление числа -5. Для наглядности проделаем смену знака ещё раз: инвертируем все биты в представлении числа -5, получаем 00000100, прибавляем единицу, получаем 00000101, то есть снова число 5, что и требовалось. Как несложно убедиться, для представления нуля операция смены знака инвариантна, то есть ноль остаётся нулём: 00000000 $\xrightarrow{inv.}$ 11111111 $\xrightarrow{+1}$ 00000000.

Такая же ситуация несколько неожиданно возникает для числа -128 (в восьмибитном случае) или, говоря вообще, для максимального по модулю отрицательного числа заданной разрядности: 100000000 $\xrightarrow{inv.}$ 01111111 $\xrightarrow{+1}$ 10000000. Это обусловлено отсутствием в данной разрядности положительного числа с таким же модулем, то есть при применении операции замены знака к комбинации 100...00 происходит *переполнение*.

§ 1.4. История платформы i386

В 1971 году корпорация Intel выпустила в свет семейство микросхем, получившее название MCS-4. Одна из этих микросхем, Intel 4004, представляла собой первый в мире законченный центральный процессор на одном кристалле, т. е., иначе говоря, первый в истории ***микропроцессор***. Машинное слово[14] этого процессора составляло четыре бита. Год спустя Intel выпустила восьмибитный процессор Intel 8008, а в 1974 году — более совершенный Intel 8080. Интересно, что 8080 использовал иные коды операций, но при этом программы, написанные на языке ассемблера для 8008, могли быть без изменений оттранслированы и для 8080. Аналогичную «совместимость по исходному коду» конструкторы Intel поддержали и для появившегося в 1978 году 16-битного процессора Intel 8086. Выпущенный годом позже процессор Intel 8088 представлял собой практически такое же устройство, отличающееся только разрядностью внешней шины (для 8088 она составляла 8 бит, для 8086 — 16 бит). Именно процессор 8088 был использован в компьютере IBM PC, давшем начало многочисленному и невероятно популярному[15] семейству

[14]Напомним, что машинным словом называется порция информации, обрабатываемая процессором в один приём.

[15]Популярность IBM-совместимых машин представляет собой явление весьма неоднозначное; многие другие архитектурные решения, имевшие существенно лучший дизайн, не смогли выжить на рынке, затопленном IBM-совместимыми компьютерами,

машин, до сих пор называемых ***IBM PC-совместимыми*** или просто ***IBM-совместимыми***.

Процессоры 8086 и 8088 не поддерживали защиты памяти и не имели разделения команд на обычные и привилегированные, так что запустить мультизадачную операционную систему на компьютерах с этими процессорами было невозможно. То же самое можно было сказать и относительно процессора 80186, выпущенного в 1982 году. В сравнении со своими предшественниками этот процессор работал гораздо быстрее за счёт аппаратной реализации некоторых операций, выполнявшихся в предыдущих процессорах путём исполнения микрокода, и за счёт повышения тактовой частоты. Процессор включал в себя некоторые подсистемы, которые ранее требовалось поддерживать с помощью дополнительных микросхем — такие как контроллер прерываний и контроллер прямого доступа к памяти. Кроме того, система команд процессора была расширена введением дополнительных команд; так, стало возможным с помощью одной команды занести в стек все регистры общего назначения. Адресная шина процессоров 8086, 8088 и 80186 была 20-разрядной, что позволяло адресовать не более 1 Mb оперативной памяти.

В том же 1982 году увидел свет и процессор 80286, ставший последним 16-битным процессором в рассматриваемом ряду. Этот процессор поддерживал так называемый защищённый режим работы (protected mode), в котором реализовывалась сегментная модель виртуальной памяти, подразумевающая, в том числе, и защиту памяти; четыре ***кольца защиты*** позволили запретить пользовательским задачам выполнение действий, влияющих на систему в целом, что необходимо при работе мультизадачной операционной системы. Адресная шина получила четыре дополнительных разряда, увеличив, таким образом, максимальное количество непосредственно доступной памяти до 16 Mb.

Однако по-настоящему мультизадачные операционные системы были реализованы лишь на следующем процессоре в ряду, 32-разрядном Intel 80386, для краткости обозначаемом просто «i386». Этот процессор, массовый выпуск которого начался в 1986 году, резко отличался от своих предшественников, прежде всего, увеличением регистров до 32 бит, существенным расширением системы команд, увеличением адресной шины до 32 разрядов, что позволяло непосредственно адресовать до 4 Gb физической памяти. Добавление поддержки ***страничной организации виртуальной памяти***, наилучшим образом пригодной для реализации мультизадачного режима работы, завершило картину. Именно с появлением i386 так называемые IBM-совместимые компьютеры, наконец, стали полноценными вычислительными системами. Вместе с тем, i386 полностью сохранил совместимость с предшествующими процессорами своей

более дешевыми за счёт их массовости. Так или иначе, в настоящее время ситуация именно такова и никаких тенденций к её изменению не предвидится.

серии, чем обусловлена достаточно странная на первый взгляд система регистров. Например, универсальные регистры процессоров 8086–80286 назывались AX, BX, CX и DX и содержали 16 бит данных каждый; в процессоре i386 и более поздних процессорах линейки имеются регистры, содержащие по 32 бита и называющиеся EAX, EBX, ECX и EDX (буква E означает слово «extended», т. е. «расширенный»), причём младшие 16 бит каждого из этих регистров сохраняют старые названия (соответственно, AX, BX, CX и DX). Большинство инструкций работает по-разному для операндов длиной 8 бит, 16 бит и 32 бита, и т. п.

Дальнейшее развитие семейства процессоров x86 вплоть до 2003 года было чисто количественным: увеличивалась скорость, добавлялись новые команды, но принципиальных изменений архитектуры не происходило. В 2003 году компания AMD представила новый процессор, имеющий 64-битные регистры, и к настоящему времени многие операционные системы способны выполняться на таких процессорах, однако наиболее популярной остаётся до сих пор именно 32-битная платформа, родоначальником которой стал процессор i386.

§ 1.5. Знакомимся с инструментом

Прежде чем написать первую самостоятельную программу на языке ассемблера, нам необходимо изучить процессор, с которым мы будем работать (пусть даже не все его возможности, но хотя бы некоторую существенную их часть), а также синтаксис языка ассемблера. К сожалению, здесь возникает определённая проблема: изучать эти две вещи одновременно не получается, но, в то же время, изучать систему команд процессора, не имея никакого представления о синтаксисе языка ассемблера, а равно и изучать синтаксис, не имея представления о системе команд — задача неблагодарная, так что, с чего бы мы ни начали, результат получится несколько странный. Мы попробуем пойти иным путём. Некоторое представление о системе команд у нас уже есть, пусть даже оно весьма и весьма слабое; попробуем получить аналогичное представление и о синтаксисе языка ассемблера, а затем уже приступим к систематическому изучению того и другого.

Сейчас мы напишем работающую программу на языке ассемблера, оттранслируем её и запустим. Поначалу в тексте программы будет далеко не всё понятно; что-то мы объясним прямо сейчас, что-то оставим до более подходящего момента. Задачу мы для себя выберем очень простую: напечатать[16] пять раз слово «Hello». Как мы уже говорили на стр. 17,

[16]Т. е. вывести на экран, или, если говорить строго, *вывести в поток стандартного вывода*; отметим, что процессор сам по себе ничего не знает о выводе на экран, все операции ввода-вывода требуют работы с внешними устройствами и организуются

для вывода строки на экран, а также для корректного завершения программы нам потребуется обращаться к операционной системе, но мы пока воспользуемся для этого уже готовыми *макросами*, которые описаны в отдельном файле. Ассемблер, сверяясь с этим файлом и с нашими указаниями, преобразует каждое использование такого макроса во фрагмент кода на языке ассемблера и сам же эти фрагменты затем оттранслирует. Поэтому в нашей программе будет очень мало мнемоник, обозначающих собственно машинные команды; в основном текст программы будет состоять из *директив*. Итак, пишем текст программы:

```
%include "stud_io.inc"
global  _start

section .text
_start: mov     eax, 0
again:  PRINT   "Hello"
        PUTCHAR 10
        inc     eax
        cmp     eax, 5
        jl      again
        FINISH
```

Попробуем теперь кое-что объяснить. Первая строчка программы содержит директиву `%include`; эта директива предписывает ассемблеру вставить на место самой директивы всё содержимое некоторого файла, в данном случае — файла `stud_io.inc`. Этот файл также написан на языке ассемблера и содержит описания макросов `PRINT`, `PUTCHAR` и `FINISH`, которые мы будем использовать для печати строки, для перехода на следующую строку на экране, а также для завершения программы. Таким образом, увидев директиву `%include`, ассемблер прочитает файл с описаниями макросов, в результате чего мы сможем их использовать.

Важно отметить, что директива `%include` обязательно должна стоять в тексте программы *раньше*, чем там встретятся имена макросов. Ассемблер просматривает наш текст сверху вниз. Изначально он ничего не знает о макросах и не сможет их обработать, если ему о них не сообщить. Просмотрев файл, содержащий описания макросов, ассемблер запоминает эти описания и продолжает их помнить до окончания трансляции, так что мы можем их использовать в программе — но не раньше, чем о них узнает ассемблер. Именно поэтому мы поставили директиву `%include` в самое начало программы: теперь макросы можно использовать во всём её тексте.

операционной системой, она же предоставляет нашей задаче абстрактные «стандартные потоки ввода-вывода».

После директивы `%include` мы видим строку со словом `global`; это тоже директива, но к ней мы вернёмся чуть подзнее.

Следующая строка программы содержит директиву `section`. Исполняемый файл в ОС Unix устроен так, что в нём машинные команды хранятся в одном месте, а инициализированные (т. е. такие, которым прямо в программе задаётся начальное значение) данные — в другом, и, наконец, в третьем месте содержится информация о том, сколько программе потребуется памяти под неинициализированные данные. В связи с этим мы должны наш исполняемый код поместить в одну «секцию», описания областей памяти с заданным начальным значением — в другую «секцию», описания областей памяти без задания начальных значений — в третью «секцию». Соответствующие секции называются `.text`, `.data` и `.bss`. В нашей простой программе мы обходимся только секцией `.text`, и рассматриваемая директива как раз и приказывает ассемблеру приступить к формированию этой секции. В будущем при рассмотрении более сложных программ нам придётся встретиться со всеми тремя секциями.

Далее в программе мы видим строку

```
_start: mov     eax, 0
```

Как мы уже знаем, словом `mov` обозначается команда, заставляющая процессор переслать некоторые данные из одного места в другое; для команды `mov` мы всегда должны указывать два *операнда*, причём первый из них будет задавать то место, *куда* следует скопировать данные, а второй операнд указывает, *какие* данные следует туда скопировать. В данном конкретном случае команда требует занести число `0` (ноль) в регистр `EAX`[17]. Значение, хранимое в регистре `EAX`, мы будем использовать в качестве счётчика цикла, то есть оно будет означать, сколько раз мы уже напечатали слово «Hello»; ясно, что в начале этот счётчик должен быть равен нулю, поскольку мы пока не напечатали ничего.

Итак, рассматриваемая строка означает приказ процессору занести ноль в `EAX`; но что за загадочное «`_start:`» в начале строки?

Слово `_start` (знак подчёркивания в данном случае является частью слова) представляет собой пример так называемых ***меток***. Попробуем сначала объяснить, что такое собой представляют эти метки «вообще», а потом расскажем, зачем нужна метка в данном конкретном случае.

[17]Читатель, уже имеющий опыт программирования на языке ассемблера, может заметить, что «правильнее» это сделать совсем другой командой: `xor eax, eax`, поскольку это позволяет достичь того же эффекта быстрее и с меньшими затратами памяти; однако для простейшего учебного примера такой трюк слишком сложен и требует неоправданно длинных пояснений. Впрочем, позже мы к этому вопросу вернёмся и обязательно рассмотрим этот и другие подобные трюки.

Команду `mov eax,0` ассемблер преобразует в некий машинный код[18], который во время выполнения программы будет находиться в какой-то области оперативной памяти (в данном случае — в пяти ячейках, идущих подряд). В некоторых случаях нам нужно знать, какой адрес будет иметь та или иная область памяти; если говорить о командах, то знать адрес нам может потребоваться, например, чтобы в какой-то момент заставить процессор произвести в это место программы условный или безусловный переход (про переходы мы уже говорили, см. стр. 11).

Конечно, мы можем использовать оперативную память и для хранения данных, а не только команд. Области памяти, предназначенные для данных, мы обычно называем ***переменными***, и даём им имена почти так же, как и в привычных нам языках программирования высокого уровня. Естественно, нам требуется знать, какой адрес имеет начало области памяти, отведённой под переменную. Адрес, как мы уже говорили, задаётся[19] числом из восьми шестнадцатеричных цифр, например, `18b4a0f0`. Запоминать такие числа нам неудобно, к тому же на момент написания программы мы ещё не знаем, в каком именно месте памяти в итоге окажется размещена та или иная команда или переменная. И здесь нам на помощь как раз и приходят метки. **Метка — это вводимое программистом слово (идентификатор), с которым ассемблер ассоциирует некоторое число, чаще всего — адрес в памяти**. В данном случае `_start` как раз и есть такая метка. Если ассемблер видит метку *перед* командой (или, как мы увидим позже, директивой, выделяющей память под переменную), он воспринимает это как указание завести в своих внутренних таблицах новую метку и связать с ней соответствующий адрес, если же метка встречается в параметрах команды, то ассемблер «вспоминает», какой именно адрес (или просто число) связано с данной меткой и подставляет этот адрес (число) вместо метки в команду. Таким образом, с меткой `_start` в нашей программе будет связано число, представляющее собой адрес ячейки, начиная с которой в оперативной памяти будет размещён машинный код, соответствующий команде `mov eax,0` (код `b8 00 00 00 00`).

Важно понимать, что метки существуют только в памяти самого ассемблера и только во время трансляции программы. Готовая к исполнению программа на машинном коде не будет содержать никаких меток, а только подставленные вместо них адреса.

---

[18]Отметим для наглядности, что машинный код этой команды состоит из пяти байтов: `b8 00 00 00 00`, первый из которых задаёт собственно действие «поместить заданное число в регистр», а также и номер регистра `EAX`. Остальные четыре байта задают (все вместе) то число, которое должно быть помещено в регистр; в данном случае это число 0.

[19]Во всяком случае, для того процессора и той системы, которые мы рассматриваем.

После метки мы поставили символ двоеточия. Интересно, что мы могли бы его и не ставить. Некоторые ассемблеры отличают метки, снабженные двоеточиями, от меток без двоеточий; но наш NASM к таким не относится. Иначе говоря, мы сами решаем, ставить двоеточие после метки или нет. Обычно программисты ставят двоеточия после меток, которыми помечены машинные команды (то есть после таких меток, куда можно передать управление), но не ставят двоеточия после меток, помечающих данные в памяти (переменные). Поскольку метка `_start` как раз и помечает команду, после неё мы двоеточие решили поставить.

Однако внимательный читатель может обратить внимание, что никаких переходов на метку `_start` в нашей программе не делается. Зачем же она тогда нужна? Дело в том, что слово «`_start`» — это специальная метка, которой помечается ***точка входа в программу***, то есть то место в программе, куда операционная система должна передать управление после загрузки программы в оперативную память; иначе говоря, метка `_start` обозначает то место, с которого начнётся выполнение программы.

Вернёмся к тексту программы и рассмотрим следующую строчку:

```
again:  PRINT   "Hello"
```

Как несложно догадаться, слово `again` в начале строки — это ещё одна метка. Слово «again» по-английски означает «снова». Дело в том, что сюда нам придётся вернуться ещё четыре раза, чтобы в итоге слово `Hello` оказалось напечатано пять раз; отсюда и название метки. Стоящее далее в строке слово `PRINT` является ***именем макроса***, а строка `"Hello"` — ***параметром*** этого макроса. Сам макрос описан, как уже говорилось, в файле `stud_io.inc`. «Увидев» имя макроса и параметр, наш ассемблер подставит вместо него целый ряд команд и директив, исполнение которых приведёт в конечном итоге к выдаче на экран строки «Hello».

Очень важно понимать, что `PRINT` не имеет никакого отношения к возможностям центрального процессора. Мы уже несколько раз упоминали этот факт, но тем не менее повторим ещё раз: `PRINT` — это не имя какой-либо команды процессора, процессор как таковой не умеет ничего печатать. Рассматриваемая нами строчка программы представляет собой не команду, а директиву, также называемую ***макровызовом***. Повинуясь этой директиве, ассемблер сформирует фрагмент текста на языке ассемблера (отметим для наглядности, что в данном случае этот фрагмент будет состоять из 23 строк в случае применения ОС Linux и из 15 строчек — для ОС FreeBSD) и сам же оттранслирует этот фрагмент, получив последовательность машинных инструкций. Эти инструкции будут содержать, в числе прочего, и обращение к операционной системе за услугой вывода данных (системный вызов `write`). Набор макросов, включающий в себя и макрос `PRINT`, введён для удобства работы на первых порах, пока мы ещё не знаем, как обращаться к операционной системе.

Позже мы узнаем это, и тогда макросы, описанные в файле stud_io.inc, станут нам не нужны; более того, мы и сами научимся создавать такие макросы.

Вернёмся к тексту нашего примера. Следующая строчка имеет вид

```
        PUTCHAR 10
```

Это тоже вызов макроса, называемого PUTCHAR и предназначенного для вывода на печать одного символа. В данном случае мы используем его для вывода символа с кодом 10; это специальный символ, обозначающий ***перевод строки***, то есть при выводе этого символа на печать курсор на экране перейдёт на следующую строку. Обратите внимание, что в этой и последующих строках присутствуют только команды и макровызовы, а меток нет. Они нам не нужны, поскольку ни на одну из последующих команд мы не собираемся делать переходы, и, значит, нам не нужна информация об адресах в памяти, где будут располагаться эти команды.

Следующая строка в программе такая:

```
        inc     eax
```

Здесь мы видим машинную команду inc, означающую приказ увеличить заданный регистр на 1. В данном случае увеличивается регистр EAX. Напомним, что в регистре EAX мы условились хранить информацию о том, сколько раз уже напечатано слово «Hello». Поскольку выполнение двух предыдущих строчек программы, содержащих вызовы макросов PRINT и PUTCHAR, привело в конечном счёте как раз к печати слова «Hello», следует отразить этот факт в регистре, что мы и делаем. Отметим, что машинный код этой команды оказывается очень коротким — всего один байт (шестнадцатеричное 40, десятичное 64).

Далее в нашей программе идёт команда сравнения:

```
        cmp     eax, 5
```

Машинная команда сравнения двух целых чисел обозначается мнемоникой cmp от английского «to compare» — сравнивать. В данном случае сравниваются содержимое регистра EAX и число 5. Результаты сравнения записываются в специальный регистр процессора, называемый ***регистром флагов***. Это позволяет, например, произвести *условный переход* в зависимости от результатов предшествующего сравнения, что мы в следующей строчке программы и делаем:

```
        jl      again
```

Здесь jl (от слов «Jump if Lower») — это мнемоника для машинной команды условного перехода, который выполняется в случае, если предшествующее сравнение дало результат «первый операнд меньше второго»,

то есть, в нашем случае, если число в регистре EAX оказалось меньше, чем 5. В терминах нашей задачи это означает, что слово «Hello» было напечатано меньше пяти раз и, стало быть, необходимо продолжать его печатать, что и делается переходом (***передачей управления***) на команду, помеченную меткой again.

Если результат сравнения был любым другим, кроме «меньше», команда jl не произведёт никаких действий, и процессор, таким образом, перейдёт к выполнению следующей по порядку команды. Это произойдёт в случае, если слово «Hello» уже было напечатано 5 раз, так что цикл пора заканчивать. После окончания цикла наша исходная задача оказывается решена, и, стало быть, программу тоже пора завершать. Для этого и предназначена следующая строка программы:

```
            FINISH
```

Слово FINISH тоже обозначает макрос; этот макрос разворачивается в последовательность команд, осуществляющих обращение к операционной системе с просьбой завершить выполнение нашей программы.

Нам осталось вернуться к началу программы и рассмотреть строку

```
    global  _start
```

Слово global — это директива, которая требует от ассемблера считать некоторую метку «глобальной», то есть как бы видимой извне (если говорить строго, видимой извне объектного модуля; это понятие мы будем рассматривать позднее). В данном случае «глобальной» объявляется метка _start. Как мы уже знаем, это специальная метка, которой помечается ***точка входа в программу***, то есть то место в программе, куда операционная система должна передать управление после загрузки программы в оперативную память. Ясно, что эта метка должна быть видна извне, что и достигается директивой global.

Итак, наша программа состоит из трёх частей: подготовки, *цикла*, начало которого отмечено меткой again, и завершающей части, состоящей из одной строчки FINISH. Перед началом цикла мы заносим в регистр EAX число 0, затем на каждой итерации цикла печатаем слово «Hello», делаем перевод строки, увеличиваем на единицу содержимое регистра EAX, сравниваем его с числом 5; если в регистре EAX всё ещё содержится число, меньшее пяти, переходим снова к началу цикла (то есть на метку again), в противном случае выходим из цикла и завершаем выполнение программы.

Чтобы попробовать приведённую программу, как говорится, в деле, необходимо войти в систему Unix, вооружиться каким-нибудь редактором текстов, набрать вышеприведённую программу и сохранить её в фай-

ле с именем, заканчивающимся[20] на `.asm` — именно так обычно называют файлы, содержащие исходный текст на языке ассемблера.

Допустим, мы сохранили текст программы в файле `hello5.asm`. Для получения исполняемого файла нам необходимо выполнить два действия. Первое — это вызов ассемблера NASM, который, используя заданный нами исходный текст, построит *объектный модуль*. Объектный модуль — это ещё не исполняемый файл; дело в том, что большие программы обычно состоят из целого набора исходных файлов, называемых ***модулями***, плюс к тому мы можем захотеть воспользоваться чьими-то сторонними подпрограммами, объединёнными в ***библиотеки***. Таким образом, нам нужно будет соединить несколько модулей воедино и подключить к ним библиотеки; этим занимается ***системный компоновщик***, также называемый иногда ***редактором связей*** или ***линкером***.

Наша примерная программа состоит всего из одного модуля и не нуждается ни в каких библиотеках, но стадии сборки (компоновки) это не исключает. Это и есть второе действие, необходимое для построения исполняемого файла: небходимо вызвать компоновщик, чтобы он нам из объектного файла построил файл исполняемый. Как раз на этой стадии будет использована метка `_start`; мы можем уточнить, что директива `global` не просто делает метку «видимой извне», а заставляет ассемблер вставить в объектный файл информацию об этой метке, видимую для компоновщика.

Итак, для начала вызываем ассемблер NASM:

```
nasm -f elf hello5.asm
```

Флажок «`-f elf`» указывает ассемблеру, что на выходе мы ожидаем объектный файл в формате ELF — именно этот формат используется в нашей системе для исполняемых файлов[21]. Результатом запуска ассемблера станет файл `hello5.o`, содержащий объектный модуль. Теперь мы можем запустить компоновщик, который называется `ld`:

```
ld hello5.o -o hello5
```

Если вы работаете под управлением 64-битной операционной системы, придётся добавить ещё один ключ для компоновщика, чтобы тот произвёл сборку 32-битного исполняемого файла; в частности, для GNU ld под Linux это будет выглядеть так:

[20]Работая в системе семейства Windows, мы, возможно, сказали бы, что `.asm` — это «расширение» файла. В ОС Unix понятие «расширения» обычно не используется, вместо него мы говорим, что имя *заканчивается на* `.asm` или что имя *имеет суффикс* `.asm`.

[21]Это верно по крайней мере для современных версий операционных систем Linux и FreeBSD. В других системах вам может потребоваться другой формат объектных и исполняемых файлов; сведения об этом обычно есть в технической документации.

```
ld -m elf_i386 hello5.o -o hello5
```

Флажком `-o` мы задали *имя исполняемого файла* (`hello5`, на этот раз без суффикса). Запустим его на исполнение, дав команду «`./hello5`». Если мы нигде не ошиблись, мы увидим пять строчек `Hello`.

§ 1.6. Макросы из файла `stud_io.inc`

Макросы, описанные в файле `stud_io.inc`, нам неоднократно потребуются в дальнейшем, поэтому, чтобы не возвращаться к ним, ещё раз приведём описание их возможностей. Текст файла `stud_io.inc` (версии для Linux и FreeBSD) приведён в приложении A, так что при желании вы легко поймёте, как устроены эти макросы. В программе, которую мы разобрали в предыдущем параграфе, мы использовали макросы `PRINT`, `PUTCHAR` и `FINISH`. Кроме этих трёх макросов наш файл `stud_io.inc` поддерживает ещё макрос `GETCHAR`, так что всего этих макросов четыре.

Макрос `PRINT` предназначен для печати строки; его аргументом должна быть строка в апострофах или двойных кавычках, ничего другого он печатать не умеет.

Макрос `PUTCHAR` предназначен для вывода на печать одного символа. В качестве аргумента он принимает код символа, записанный в виде числа или в виде самого символа, взятого в кавычки или апострофы; также можно в качестве аргумента этого макроса использовать однобайтовый регистр — `AL`, `AH`, `BL`, `BH`, `CL`, `CH`, `DL` или `DH`. **Использовать другие регистры в качестве аргумента `PUTCHAR` нельзя!** Наконец, аргументом этого макроса может выступать исполнительный адрес, заключённый в квадратные скобки — в этом случае код символа будет взят из ячейки памяти по этому адресу.

Макрос `GETCHAR` считывает символ из потока стандартного ввода (с клавиатуры). После считывания код символа записывается в регистр `EAX`; поскольку код символа всегда умещается в один байт, его можно извлечь из регистра `AL`, остальные разряды `EAX` будут равны нулю. Если символов больше нет (достигнута так называемая ***ситуация конца файла***, которая в ОС Unix обычно имитируется нажатием Ctrl-D), в `EAX` будет занесено значение `-1` (шестнадцатеричное `FFFFFFFF`, то есть все 32 разряда регистра равны единицам). Никаких параметров этот макрос не принимает.

Макрос `FINISH` завершает выполнение программы. Этот макрос можно вызвать без параметров, а можно вызвать с одним числовым параметром, задающим так называемый ***код завершения процесса***; обычно используют код `0`, если наша программа отработала успешно, и код `1`, если в процессе работы возникли ошибки.

Глава 2. Процессор i386

§ 2.1. Система регистров i386

Регистром называют электронное устройство в составе центрального процессора, способное содержать в себе определённое количество данных в виде двоичных разрядов. В большинстве случаев (но не всегда) содержимое регистра трактуется как целое число, записанное в двоичной системе счисления. Регистры процессора i386 можно условно разделить на ***регистры общего назначения***, ***сегментные регистры*** и ***специальные регистры***. Каждый регистр имеет своё название[1], состоящее из двух-трёх латинских букв.

Сегментные регистры (CS, DS, SS, ES, GS и FS) в «плоской» модели памяти не используются. Точнее говоря, перед передачей управления пользовательской задаче операционная система заносит в эти регистры некоторые значения, которые задача теоретически может изменить, но ничего хорошего из этого всё равно не выйдет — скорее всего, произойдёт аварийное завершение. Таким образом, мы принимаем во внимание существование этих регистров, но более к ним возвращаться не будем.

Регстры общего назначения процессора i386 — это 32-битные регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP. Как уже отмечалось на стр. 24, буква E в названии этих регистров означает слово «extended», подчёркивая тот факт, что в их современном виде эти регистры появились только в процессоре i386. Для совместимости с предыдущими процессорами семейства x86 каждый 32-битный регистр имеет обособленную младшую половину (младшие 16 бит), имеющую отдельное название, получаемое отбрасыванием буквы E, то есть, иначе говоря, мы можем работать также с 16-битными регистрами AX, BX, CX, DX, SI, DI, BP и SP, которые представляют собой младшие половины соответствующих 32-битных регистров.

[1] Этим процессоры семейства x86 отличаются от многих других процессоров, в которых регистры имеют номера.

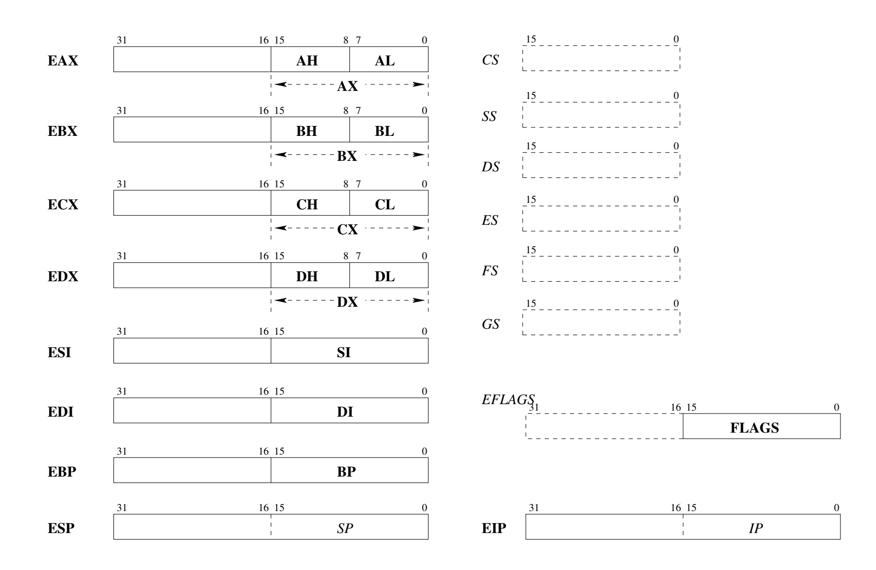


Рис. 2.1. Система регистров i386

Кроме того, регистры AX, BX, CX и DX также делятся на младшие и старшие части, теперь уже восьмибитные. Так, для регистра AX его младший байт имеет также название AL, а старший байт — AH (от слов «low» и «high»). Аналогично мы можем работать с регистрами BL, BH, CL, CH, DL и DH, которые представляют собой младшие и старшие байты регистров BX, CX и DX. Остальные регистры общего назначения таких обособленных однобайтовых подрегистров не имеют.

Каждый из регистров общего назначения, несмотря на такое название, в некоторых случаях играет специфическую, только ему присущую роль, частично закодированную в имени регистра. Так, в имени регистра AX буква A обозначает слово «accumulator»; на многих архитектурах, включая знаменитый IAS Джона фон Неймана, ***аккумулятором*** называли регистр, участвующий (по определению) во всех арифметических операциях, во-первых, в качестве одного из операндов, и, во-вторых, в качестве места, куда следует поместить результат. Связанная с этим особая роль регистров AX и EAX проявляется в командах целочисленного умножения и деления (см. § 2.3.4).

В имени регистра BX буква B обозначает слово «base», но никакой особой роли в 32-битных процессорах этому регистру не отведено (хотя в 16-битных процессорах такая роль существовала).

В имени CX буква C обозначает слово «counter» (счётчик). Регистры ECX, CX, а в некоторых случаях даже CL используются во многих машинных командах, предполагающих (в том или ином смысле) определённое количество итераций.

Имя регистра DX символизирует слово «data» (данные). В особой роли регистр EDX (или DX, если выполняется шестнадцатиразрядная операция) выступает при выполнении операций целочисленного умножения (для хранения части результа-

та, не поместившейся в аккумулятор) и целочисленного деления (для хранения старшей части делимого, а после выполнения операции — для хранения остатка от деления).

Имена регистров SI и DI означают, соответственно, «source index» и «destination index» (индекс источника и индекс назначения). Регистры ESI и EDI используются в командах, работающих с массивами данных, причём ESI хранит адрес текущей позиции в массиве-источнике (например, в области памяти, которую нужно куда-то скопировать), а EDI хранит адрес текущей позиции в массиве-цели (в области памяти, *куда* производится копирование).

Имя регистра BP обозначает «base pointer» (базовый указатель). Как правило, регистр EBP используется для хранения базового адреса стекового фрейма при вызове подпрограмм, имеющих параметры и локальные переменные.

Наконец, имя регистра SP обозначает «stack pointer» (указатель стека). Несмотря на принадлежность регистра ESP к группе регистров общего назначения, в реальности он всегда используется именно в качестве указателя стека, то есть хранит адрес текущей позиции вершины аппаратного стека. Поскольку обойтись без стека тяжело, а другие регистры для этой цели не подходят, можно считать, что ESP никогда не выступает ни в какой иной роли.

К регистрам специального назначения мы отнесём ***регистр счётчика команд*** EIP и ***регистр флагов*** FLAGS.

Регистр EIP, имя которого образовано от слов «extended instruction pointer», хранит в себе *адрес в оперативной памяти, по которому процессору следует извлечь следующую машинную инструкцию, предназначенную к выполнению.* После извлечения инструкции из памяти значение в регистре EIP автоматически увеличивается на длину прочитанной инструкции (отметим, что инструкция может занимать в памяти от одной до одиннадцати идущих подряд ячеек), так что регистр снова содержит адрес команды, которую нужно выполнить следующей. Как и для регистров общего назначения, младшая половина регистра EIP имеет имя IP, однако использовать его, работая под управлением 32-битной операционной системы, мы всё равно никак не сможем.

Регистр флагов FLAGS — единственный из рассматриваемых нами регистров, который очень редко используется как единое целое, и вовсе никогда не рассматривается как число. Вместо этого каждый двоичный разряд (бит) этого регистра представляет собой ***флаг***, имеющий собственное имя. Некоторые из этих флагов процессор сам устанавливает в ноль или единицу в зависимости от результата очередной выполненной команды; другие флаги устанавливаются в явном виде соответствующими инструкциями и в дальнейшем влияют на ход выполнения некоторых команд. В частности, флаги используются для выполнения ***условных переходов***: некоторая команда выполняет арифметическую или другую операцию, а следующая прямо за ней команда передаёт управление в другое место программы, но только в случае, если результат предыду-

щей операции удовлетворяет тем или иным условиям; условия как раз и проверяются по установленным флагам. Перечислим некоторые флаги:

- `ZF` — флаг нулевого результата (zero flag). Этот флаг устанавливается в ходе выполнения арифметических операций и операций сравнения: если в результате операции получился ноль, `ZF` устанавливается в единицу.

- `CF` — флаг переноса (carry flag). После выполнения арифметической операции над беззнаковыми числами этот флаг выставляется в единицу, если потребовался перенос из старшего разряда, то есть результат не поместился в регистр, либо потребовался заём из несуществующего разряда при вычитании, то есть вычитаемое оказалось больше, чем уменьшаемое (см. § 1.3.1). В противном случае флаг выставляется в ноль.

- `SF` — флаг знака (sign flag). Устанавливается равным старшему биту результата, который для знаковых чисел соответствует знаку числа (см. стр. 21).

- `OF` — флаг переполнения (overflow flag). Выставляется в единицу, если при работе со знаковыми числами произошло переполнение (см. стр. 21).

- `DF` — флаг направления (direction flag). Этот флаг можно установить командой `STD` и обнулить командой `CLD`; в зависимости от его значения *строковые операции*, которые мы будем рассматривать несколько позже, выполняются в прямом или в обратном направлении.

- `PF` и `AF` — флаг чётности (parity flag) и флаг полупереноса (auxiliary carry flag). Нам эти флаги не потребуются.

- `IF` и `TF` — флаги разрешения прерываний (interrupt flag) и ловушки (trap flag). Эти флаги нам *недоступны*, их можно изменять только в привилегированном режиме.

На самом деле такой набор флагов существовал до процессора i386; при переходе к процессору i386 регистр флагов, как и все остальные регистры, увеличился в размерах и превратился в регистр `EFLAGS`, но все новые флаги нам в ограниченном режиме недоступны, так что рассматривать их мы не будем.

§ 2.2. Память, регистры и команда `mov`

§ 2.2.1. Память пользовательской задачи. Секции

Ясно, что регистров центрального процессора заведомо не хватит для хранения всей информации, нужной в любой более-менее сложной программе. Поэтому регистры используются лишь для краткосрочного хранения промежуточных результатов, которые вот-вот понадобятся снова. Кроме регистров, программа может воспользоваться для хранения информации *оперативной памятью.*

Один из основополагающих принципов, определяющих архитектуру фон Неймана, состоит в ***однородности памяти***: и сама программа (то есть составляющие её машинные команды), и все данные, с которыми она работает, располагаются в ячейках памяти, одинаковых по своему устройству и имеющих адреса из единого адресного пространства. В нашем случае каждая ячейка памяти способна хранить ровно один байт и имеет свой уникальный *адрес* — число из 32 бит (речь идёт, естественно, о виртуальных адресах, которые мы обсуждали на стр. 16).

Несмотря на то, что физически все ячейки памяти абсолютно одинаковы, операционная система может установить для пользовательской задачи разные возможности по доступу к различным областям памяти. Это достигается средствами аппаратной защиты памяти, которые мы уже упоминали. В частности, некоторые области памяти могут быть доступны задаче только для чтения, но не для изменения находящейся там информации; кроме того, не всякую область памяти разрешается рассматривать как машинный код (то есть заносить адреса ячеек из этой области в регистр счётчика команд). Если задаче позволено рассматривать содержимое области памяти как фрагмент исполняемой машинной программы, говорят, что область памяти ***доступна на исполнение***; область памяти, содержимое которой задача может модифицировать, называют ***доступной на запись***. Часто можно встретить также термин ***доступ на чтение***, но в применении к оперативной памяти отсутствие этого вида доступа обычно означает отсутствие какого-либо доступа вообще.

Обычно современные операционные системы выстраивают виртуальное адресное пространство пользовательской задачи, разделив его на четыре основные ***секции***. Первая из этих секций, называемая ***секцией кода***, создаётся для хранения исполняемого машинного кода, из которого, собственно говоря, и состоит исполняемая программа. Естественно, область памяти, выделенная под секцию кода, доступна задаче на исполнение. С другой стороны, **операционная система не позволяет пользовательским задачам модифицировать содержимое секции кода**; попытка задачи сделать такую модификацию рассматривается как

нарушение защиты памяти. Сделано это по достаточно простой причине: если в системе одновременно запущено в виде задач несколько экземпляров одной и той же программы, операционная система обычно хранит в физической памяти только один экземпляр машинного кода такой программы. Это верно даже в случае, если запущенные задачи принадлежат разным пользователям и имеют разные полномочия в системе. Если одна из таких задач модифицирует «свою» секцию кода, очевидно, что это помешает работать остальным — ведь они используют (физически) ту же самую секцию кода. Однако на чтение секция кода доступна, так что её можно использовать не только для кода как такового, но и для хранения *константных данных* — такой информации, которая не изменяется во время выполнения программы. В программах секция кода обозначается «`.text`» (точка перед названием секции обязательна и является частью названия).

Вторая и третья секции, имеющие собирательное название ***сегмент данных***, предназначены для хранения глобальных и динамических переменных. Обе эти секции доступны задаче как на чтение, так и на запись; с другой стороны, операционная система обычно запрещает передачу управления внутрь этих секций, чтобы несколько затруднить «взлом» компьютерных программ. Первая из двух секций называется собственно ***секцией данных***, в программах обозначается «`.data`[2]» и содержит *инициализированные данные*, то есть такие глобальные переменные, для которых в программе задано начальное значение. Вторая секция из сегмента данных называется ***секцией неинициализированных данных*** или ***секцией BSS***[3] и обозначается «`.bss`»; как ясно из названия, эта секция предназначена для переменных, для которых начальное значение не задано. Секция BSS отличается от секции данных двумя особенностями. Во-первых, поскольку содержимое секции данных на момент старта программы должно быть таким, как это задано программой, её образ необходимо хранить в исполняемом файле программы; для секции BSS в исполняемом файле достаточно хранить только размер. Во-вторых, секция BSS может во время работы программы увеличиваться в размерах, что позволяет создавать новые переменные на этапе выполнения.

Память, получаемую во время работы программы, обычно называют ***динамической памятью*** или ***кучей*** (heap). В нашем курсе мы не будем рассматривать работу с динамической памятью, но для любознательных читателей сообщим, что в ОС Linux выделение дополнительной памяти производится системным вызовом `brk`, о котором можно узнать из технической документации по ядру. Выделение дополнительной памяти в ОС FreeBSD производится средствами системного вы-

[2]Data (`англ.`) — данные; читается «дэйта».

[3]Исторически аббревиатура BSS обозначала Block Started by Symbol, что было обусловлено особенностями одного старого ассемблера. В настоящее время программисты предпочитают расшифровывть BSS как Blank Static Storage.

зова `mmap`, который, к сожалению, гораздо сложнее, особенно для использования в программах на языке ассемблера.

Четвёртая основная секция — это так называемая ***секция стека***; она нужна для хранения локальных переменных в подпрограммах и адресов возврата из подпрограмм. Подробный рассказ о стеке у нас ещё впереди, пока мы только отметим, что эта секция также доступна на запись; доступность её на исполнение зависит от конкретной операционной системы и даже от конкретной версии ядра: например, в большинстве версий Linux в секцию стека можно передавать управление, но специальный «патч» к исходным текстам ядра эту возможность устраняет. Эта секция также может увеличиваться в размерах по мере необходимости, причём это происходит автоматически (в отличие от увеличения секции BSS, которое необходимо затребовать от операционной системы явно). Секция стека присутствует в пользовательской задаче всегда, её исходное содержимое зависит только от параметров запуска программы, а дальше она изменяется по мере необходимости. Никакой информации о секции стека исполняемый файл не содержит. Во время написания программы мы не можем никак повлиять на секцию стека, так что ассемблер не имеет никакого специального обозначения для неё.

§ 2.2.2. Директивы для отведения памяти

Содержимое этого параграфа не имеет прямого отношения к архитектуре процессора i386; здесь мы рассмотрим директивы, являющиеся особенностью конкретного ассемблера. Дело, однако, в том, что нам очень сложно будет обойтись без них при изучении дальнейшего материала.

Написанные нами условные обозначения машинных команд ассемблер транслирует в некий *образ области памяти* — массив чисел (данных), которые нужно будет записать в смежные ячейки оперативной памяти. Затем при запуске программы в эту область памяти будет передано управление (то есть, попросту говоря, адрес какой-то из этих ячеек будет записан в регистр `EIP`) и центральный процессор начнёт ***выполнение*** нашей программы, используя числа из созданного ассемблером образа в качестве кодов команд. Если мы пишем программу, которая будет потом выполняться в качестве задачи под управлением многозадачной операционной системы, то при загрузке исполняемого файла в память операционная система сформирует секцию кода (секцию `.text`) соответствующего размера и именно в ней расположит машинный код нашей программы, то есть, попросту, скопирует в неё записанный в исполняемом файле образ памяти.

Аналогично можно использовать ассемблер и для создания образа области памяти, содержащей данные, а не команды. Для этого нужно сообщить ассемблеру, сколько памяти нам необходимо под те или иные

нужды, и при этом, возможно, задать те значения, которые в эту память будут помещены перед стартом программы.

Пользуясь нашими указаниями, ассемблер соответствующим образом сформирует отдельно образ памяти, содержащий команды (образ секции `.text`), и отдельно образ памяти, содержащий инициализированные данные (образ секции `.data`), а кроме того, сосчитает, сколько нам нужно такой памяти, о начальном значении которой мы не беспокоимся и для которой, соответственно, не нужно формировать образ, а нужно лишь указать её количество (размер секции `.bss`). Всё это ассемблер запишет в файл с объектным кодом, а системный компоновщик из таких файлов (возможно, нескольких) сформирует исполняемый файл, содержащий (кроме собственно машинного кода), во-первых, те данные, которые нужно записать в память перед стартом программы, и, во-вторых, указания на то, сколько программе понадобится ещё памяти, кроме той, что нужна под размещение машинного кода и исходных данных. Чтобы сообщить ассемблеру, в какой секции должен быть размещён тот или иной фрагмент формируемого образа памяти, мы в программе на языке ассемблера должны использовать директиву `section`; например, строчка

```
section .text
```

означает, что результат обработки последующих строк должен размещаться в секции кода, а строчка

```
section .bss
```

заставляет ассемблер перейти к формированию секции неинициализированных данных. Директивы переключения секций могут встречаться в программе сколько угодно раз — мы можем сформировать часть одной секции, затем часть другой, потом вернуться к формированию первой.

Сообщить ассемблеру о наших потребностях в оперативной памяти можно с помощью ***директив резервирования памяти***, которые делятся на два вида: директивы резервирования неинициализированной памяти и директивы задания исходных данных. Обычно перед директивами обоих видов ставится метка, чтобы можно было ссылаться с её помощью на адрес в памяти, где ассемблер отвёл для нас требуемые ячейки.

Директивы резервирования неинициализированной памяти сообщают ассемблеру, что необходимо зарезервировать заданное количество ячеек памяти, причём ничего, кроме количества, не уточняется. Мы не требуем от ассемблера заполнять отведённую память какими-либо конкретными значениями, нам достаточно, чтобы эта память вообще была в наличии. Для резервирования заданного количества однобайтовых ячеек используется директива `resb`, для резервирования па-

мяти под определённое количество «слов[4]», то есть *двухбайтовых* значений (например, коротких целых чисел) — директива `resw`, для «двойных слов» (то есть четырёхбайтных значений) используется `resd`; после директивы указывается (в качестве параметра) число, обозначающее количество значений, под которое мы резервируем память. Как уже говорилось, обычно перед директивой резервирования памяти ставится метка. Например, если мы напишем следующие строки:

```
string  resb 20
count   resw 256
x       resd 1
```

то по адресу, связанному с меткой `string`, будет расположен массив из 20 однобайтовых ячеек (такой массив можно, например, использовать для хранения строки символов); по адресу `count` ассемблер отведёт массив из 256 двубайтных «слов» (т. е. 512 ячеек), которые можно использовать, например, для каких-нибудь счётчиков; наконец, по адресу `x` будет располагаться одно «двойное слово», то есть четыре байта памяти, которые можно использовать для хранения достаточно большого целого числа.

Директивы второго типа, называемые ***директивами задания исходных данных***, не просто резервируют память, а указывают, какие значения в этой памяти должны находиться к моменту запуска программы. Соответствующие значения указываются после директивы через запятую; памяти отводится столько, сколько указано значений. Для задания однобайтовых значений используется директива `db`, для задания «слов» — директива `dw` и для задания «двойных слов» — директива `dd`. Например, строка

```
fibon   dw 1, 1, 2, 3, 5, 8, 13, 21
```

зарезервирует память под восемь двубайтных «слов» (то есть всего 16 байт), причём в первые два «слова» будет занесено число 1, в третье слово — число два, в четвёртое — число 5 и т. д. С адресом первого байта отведённой и заполненной таким образом памяти будет ассоциирована метка `fibon`.

Числа можно задавать не только в десятичном виде, но и в шестнадцатеричном, восьмеричном и двоичном. Шестнадцатеричное число в ассемблере NASM можно задать тремя способами: прибавив в конце числа букву `h` (например, `2af3h`), либо написав перед числом символ `$` (`$2af3`),

[4]Напомним, что такая терминология не совсем корректна, поскольку термином «слово» должна обозначаться порция информации, обрабатываемая процессором за один приём; начиная с i386, размер машинного слова на этих процессорах составлял четыре байта, а не два. Использование термина `word` в ассемблерах для обозначения двухбайтовых значений — пережиток тех времён, когда машинное слово составляло два байта.

либо поставив перед числом символы 0x, как в языке Си (0x2af3). При использовании символа $ необходимо следить, чтобы сразу после $ стояла цифра, а не буква, так что если число начинается с буквы, необходимо добавить 0 (например, $0f9 вместо просто $f9). Аналогично нужно следить за первым символом и при использовании буквы h: например, a21h ассемблер воспримет как идентификатор, а не как число. Чтобы избежать проблемы, следует написать 0a21h. С другой стороны, с числом 2fah такой проблемы изначально не возникает, поскольку первый символ в его записи является цифрой. Восьмеричное число обозначается добавлением после числа буквы o или q (например, 634o, 754q). Наконец, двоичное число обозначается буквой b (10011011b).

Отдельного упоминания заслуживают коды символов и *текстовые строки*. Для работы с текстовыми данными каждому символу приписывается ***код символа*** — небольшое целое положительное число. Таблица, ставящая каждому символу в соответствие его код, называется ***кодировкой*** символов. Все современные компьютерные системы используют кодировку ASCII для представления латинских букв, а также цифр, знаков препинания и некоторых других символов. Например, код заглавной латинской буквы «A» в кодировке ASCII равен 65, код цифры «0» (ноль) — число 48, код знака «+» (плюс) — 43, а код пробела — 32. Текстовые данные могут содержать также «специальные символы», которые не отображаются в виде символов, а обозначают свойства текста; например, символ с кодом 10 обозначает перевод строки, то есть при его выводе на экран курсор на экране перейдёт на следующую строку. Кодировка ASCII использует числа от 1 до 127, так что для хранения одного символа оказывается заведомо достаточно одной однобайтовой ячейки памяти[5]. Для хранения строк символов обычно используются массивы однобайтовых ячеек, в каждой из которых содержится код очередного символа.

Чтобы программисту не нужно было запоминать коды, соответствующие печатным символам (буквам, цифрам и т. п.), вместо кода можно написать сам символ, взяв его в апострофы или двойные кавычки. Так, директива

```
fig7    db '7'
```

[5]Отметим, что в таблицу ASCII не входят буквы никаких алфавитов, кроме латинского — ни русские (кириллические) буквы, ни греческие, ни даже латинские буквы с диакритическими знаками, такие как немецкая «ä» или шведская å, не имеют своего кода в ASCII-таблице. К представлению символов, не вошедших в ASCII, возможно много различных подходов: иногда их кодируют числами от 128 до 255, что позволяет по-прежнему уместить каждый символ в один байт, но не позволяет сочетать несколько разных алфавитов (например, кириллица вместе с греческими буквами в отведённое пространство кодов не поместятся, не говоря уже об иероглифах); иногда (особенно в последние годы) используют многобайтные кодировки, в которых один символ может занимать два, три или четыре байта.

разместит в памяти байт, содержащий число 55 — код символа «семёрки», а адрес этой ячейки свяжет с меткой `fig7`. Мы можем написать и сразу целую строку, например, вот так:

```
welmsg  db 'Welcome to Cyberspace!'
```

В этом случае по адресу `welmsg` будет располагаться строка из 16 символов (то есть массив однобайтовых ячеек, содержащих коды соответствующих символов). Как уже было сказано, кавычки можно использовать как одинарные (апострофы), так и двойные, так что следующая строка полностью аналогична предыдущей:

```
welmsg  db "Welcome to Cyberspace!"
```

Внутри двойных кавычек апострофы рассматриваются как обычный символ; то же самое можно сказать и о символе двойных кавычек внутри одинарных. Например, фразу «`So I say: "Don't panic!"`» можно задать следующим образом:

```
panic   db 'So I say: "Don', "'", 't panic"'
```

Здесь мы сначала воспользовались апострофом в качестве символа одинарных кавычек, так что символ двойных кавычек, обозначающий прямую речь, вошел в нашу строку как обычный символ. Затем, когда нам в строке потребовался апостроф, мы закрыли одинарные кавычки и воспользовались двойными, чтобы набрать символ апострофа. Наконец, мы снова воспользовались апострофами, чтобы задать остаток нашей фразы, включая и заканчивающий прямую речь символ двойных кавычек.

Отметим, что строками в одинарных и двойных кавычках можно пользоваться не только с директивой `db`, но и с директивами `dw` и `dd`, однако при этом необходимо учитывать некоторые тонкости, которые мы рассматривать не будем.

При написании программ обычно директивы задания исходных данных располагают в секции `.data` (то есть перед описанием данных ставят директиву `section .data`), а директивы резервирования памяти выделяют в секцию `.bss`. Это обусловлено уже упоминавшимся различием в их природе: инициализированные данные нужно хранить в исполняемом файле, тогда как для неинициализированных достаточно указать их общее количество. Секция `.bss`, как мы помним, как раз и отличается от `.data` тем, что в исполняемом файле от неё хранится только указание размера; иначе говоря, размер исполняемого файла не зависит от размера секции `.bss`. Так, если мы добавим в секцию `.data` директиву

```
        db "This is a string"
```

то размер исполняемого файла увеличится на 16 байт (надо же где-то хранить строку `"This is a string"`), тогда как если мы добавим в секцию `.bss` директиву

```
            resd 16
```

размер исполняемого файла вообще никак не изменится, несмотря на то, что памяти выделяется ровно столько же.

Расположить директивы задания исходных данных мы можем и в секции кода (секции `.text`), нужно только помнить, что тогда эти данные нельзя будет изменить во время работы программы. Но если в нашей программе есть большой массив, который не нужно изменять (какая-нибудь таблица констант, а чаще — некий текст, который наша программа должна напечатать), выгоднее разместить эти данные именно в секции кода, ведь если пользователи запустят одновременно много экземпляров нашей программы, секция кода у них будет одна на всех и мы сэкономим память. Ясно, что такая экономия возможна только для неизменяемых данных. Помните, что попытка изменить во время выполнения содержимое секции кода приведёт к аварийному завершению программы!

Ассемблер позволяет использовать любые команды и директивы в любых секциях. В частности, мы можем в секцию данных поместить машинные команды, и они будут, как обычно, оттранслированы в соответствующий машиный код, но передать управление на этот код мы не сможем. Всё же в некоторых экзотических случаях такое может иметь смысл, поэтому ассемблер молча выполнит наши указания. Встретив директивы резервирования памяти (`resb`, `resw` и др.) в секции `.data`, ассемблер тоже сделает своё дело, но в этом случае будет всё же выдано предупреждающее сообщение; действительно, ситуация несколько странная, поскольку без всякого толка увеличивает размер исполняемого файла, хотя и не приводит ни к каким фатальным последствиям. Ещё более странно будут выглядеть директивы резервирования неинициализированной памяти в секции кода: действительно, если начальное значение не задано, а изменить эту память мы не можем — значит, никакое осмысленное значение в такую память никогда не попадёт, и какой в таком случае от неё толк?! Тем не менее, ассемблер и в этом случае продолжит трансляцию, выдав только предупреждающее сообщение. Предупреждение будет выдано также и в случае, если в секции BSS встретится что-нибудь кроме директив резервирования неинициализированной памяти: ассемблер точно знает, что сформированный для этой секции образ ему будет некуда записывать. Несмотря на то, что во всех перечисленных случаях ассемблер, выдав предупреждение, продолжает работу, правильнее будет предположить, что вы ошиблись, и исправить программу.

§ 2.2.3. Команда `mov`

Одна из самых часто встречающихся в программах на языке ассемблера команд — это команда пересылки данных из одного места в другое. Она называется `mov` (от слова «move»). Для нас эта команда интересна ещё и тем, что на её примере можно обсудить целый ряд очень важных вопросов, таких как виды операндов, понятие длины операнда, прямую и косвенную адресацию, общий вид исполнительного адреса, научиться работать с метками и т. д.

Итак, команда `mov` имеет два *операнда*, т. е. два параметра, записываемых после мнемокода команды (в данном случае — слова «`mov`») и задающих объекты, над которыми команда будет работать. Первый операнд задаёт то место, *куда* будут помещены данные, а второй операнд — то, *откуда* данные будут взяты. Так, например, уже знакомая нам по вводным примерам инструкция

```
mov eax, ebx
```

копирует данные из регистра EBX в регистр EAX. Важно отметить, что **команда `mov` только копирует данные, не выполняя никаких преобразований**. Для любых преобразований следует воспользоваться другими командами, имеющими соответствующее предназначение.

§ 2.2.4. Виды операндов

В примерах, рассматривавшихся выше, мы встречали по меньшей мере два варианта использования команды `mov`:

```
mov eax, ebx
mov ecx, 5
```

Первый вариант копирует содержимое одного регистра в другой регистр, тогда как второй вариант *заносит в регистр некоторое число, заданное непосредственно в самой команде* (в данном случае число 5). На этом примере наглядно видно, что *операнды бывают разных видов*. Если в роли операнда выступает название регистра, то говорят о ***регистровом операнде***; если же значение указано прямо в самой команде, такой операнд называется ***непосредственным операндом***.

На самом деле, в рассматриваемом случае следует говорить даже не о различных типах операндов, а о *двух разных командах*, которые просто обозначаются одинаковой мнемоникой. Две команды `mov` из нашего примера переводятся в совершенно разные машинные коды, причём первая из них занимает в памяти два байта, а вторая — пять, четыре из которых тратятся на размещение непосредственного операнда.

Кроме непосредственных и регистровых операндов, существует ещё и третий вид операнда — ***адресный операнд***, называемый также операндом типа «память». В этом случае операнд задаёт (тем или иным способом) *адрес ячейки или области памяти, с которой надлежит произвести заданное командой действие*. Необходимо помнить, что **в языке ассемблера NASM операнд типа «память» абсолютно всегда обозначается квадратными скобками**, в которых и пишется собственно адрес. В простейшем случае адрес задаётся *в явном виде*, то есть в форме числа; обычно при программировании на языке ассемблера вместо чисел мы, как уже говорилось, используем метки. Например, мы можем написать:

```
section .data
; ...
count   dd 0
```

(символ «;» задаёт а языке ассемблера комментарий), описав *область памяти размером в 4 байта, с адресом которой связана метка* `count`*, и в которой исходно хранится число 0*. Если теперь написать

```
section .text
; ...
        mov [count], eax
```

эта команда `mov` будет обозначать копирование данных из регистра `EAX` в область памяти, помеченную меткой `count`, а, например, команда

```
        mov edx, [count]
```

будет, наоборот, обозначать копирование из памяти по адресу `count` в регистр `EDX`.

Чтобы понять, зачем нужны квадратные скобки, рассмотрим команду

```
        mov edx, count
```

Вспомним, что метку (в данном случае `count`), как мы уже говорили на стр. 27, ассемблер просто заменяет на некоторое *число*, в данном случае — адрес области памяти. Например, если область памяти `count` расположена в ячейках, адреса которых начинаются с `40f2a008`, то вышеприведённая команда — это абсолютно то же самое, как если бы мы написали

```
        mov edx, 40f2a008h
```

Теперь очевидно, что это просто уже знакомая нам форма команды `mov` с непосредственным операндом, т. е. эта команда *заносит в регистр EDX число 40f2a008*, не вникая в то, является ли это число адресом какой-либо ячейки памяти или нет. Если же мы добавим квадратные скобки, речь пойдёт уже об *обращении к памяти* по заданному адресу, то есть число будет использовано как адрес области памяти, где размещено значение, с которым надо работать (в данном случае поместить в регистр `EDX`).

§ 2.2.5. Прямая и косвенная адресация

Задать адрес области памяти в виде числа или метки возможно не всегда. Во многих случаях нам приходится тем или иным способом *вычислять* адрес, и уже затем обращаться к области памяти по такому вычисленному адресу. Например, именно так будут обстоять дела, если нам

потребуется заполнить все элементы какого-нибудь массива заданными значениями: адрес начала массива нам наверняка известен, но нужно будет организовать цикл (по элементам массива) и на каждом шаге цикла выполнять копирование заданного значения в *очередной* (каждый раз другой) элемент массива. Самый простой способ исполнить это — перед входом в цикл задать некий адрес равным адресу начала массива и на каждой итерации увеличивать его.

Важное отличие от простейшего случая, рассмотренного в предыдущем параграфе, состоит в том, что *адрес, используемый для доступа к памяти, будет вычисляться во время исполнения программы*, а не задаваться при её написании. Таким образом, вместо указания процессору «обратись к области памяти по такому-то адресу» нам нужно потребовать действия более сложного: «возьми там-то (например, в регистре) значение, используй это значение в качестве адреса и по этому адресу обратись к памяти». Такой способ обращения к памяти называют ***косвенной адресацией*** (в отличие от ***прямой адресации***, при которой адрес задаётся явно).

Процессор i386 позволяет для косвенной адресации использовать только значения, хранимые в регистрах процессора. Простейший вид косвенной адресации — это обращение к памяти по адресу, хранящемуся в одном из регистров общего назначения. Например, команда

```
mov ebx, [eax]
```

означает «возьми значение в регистре `EAX`, используй это значение в качестве адреса, по этому адресу обратись к памяти, возьми оттуда 4 байта и занеси эти 4 байта в регистр `EBX`», тогда как команда

```
mov ebx, eax
```

означала, как мы уже видели, просто «скопируй содержимое регистра `EAX` в регистр `EBX`».

Рассмотрим небольшой пример. Пусть у нас есть массив из однобайтовых элементов, предназначенный для хранения строки символов, и нам необходимо в каждый элемент этого массива занести код символа `'@'`. Посмотрим, с помощью какого фрагмента кода мы можем это сделать (воспользуемся командами, уже знакомыми нам из примера на стр. 25)[6]

[6]Здесь и далее комментарии к текстам примеров приводятся на русском языке. Это допустимо в учебном пособии, исходя из соображений наглядности. Следует, однако, учитывать, что в практическом программировании наличие кириллических символов в тексте программы представляет собой пример крайне плохого стиля. Комментарии в программах следует писать по-английски, что позволит любому программисту в мире прочитать текст вашей программы.

```
section .bss
array   resb 256        ; массив размером 256 байт

section .text
; ...
        mov ecx, 256    ; кол-во элементов -> в счётчик (ECX)
        mov edi, array  ; адрес массива -> в EDI
        mov al, '@'     ; нужный код -> в однобайтовый AL
again:  mov [edi], al   ; заносим код в очередной элемент
        inc edi         ; увеличиваем адрес
        dec ecx         ; уменьшаем счётчик
        jnz again       ; если там не ноль, повторяем цикл
```

Здесь мы использовали регистр ECX для хранения числа итераций цикла, которые ещё осталось выполнить (изначально 256, на каждой итерации уменьшаем на единицу, а достигнув нуля — заканчиваем цикл), а для хранения адреса мы воспользовались регистром EDI, в который перед входом в цикл занесли адрес начала массива array, а на каждой итерации увеличивали его на единицу, переходя, таким образом, к следующей ячейке.

Внимательный читатель может заметить, что фрагмент кода написан не совсем рационально. Во-первых, можно было бы использовать лишь один изменяемый регистр, либо сравнивая его не с нулём, а с числом 256, либо просматривая массив с конца. Во-вторых, не совсем понятно, зачем для хранения кода символа использовался регистр AL, ведь можно было использовать непосредственный операнд прямо в команде, заносящей значение в очередной элемент массива.

Всё это действительно так, но для этого нам пришлось бы воспользоваться, во-первых, явным указанием размера операнда, а это мы ещё не обсуждали; и, во вторых, пришлось бы использовать команду cmp, либо усложнить команду присваивания начального значения адреса. Таким образом, причина применения нами такого нерационального кода здесь — желание ограничиться наименьшим количеством пояснений, отвлекающих внимание от основной задачи.

§ 2.2.6. Общий вид исполнительного адреса

Как видно из предыдущего параграфа, адрес для обращения к памяти не всегда задан заранее; мы можем вычислить адрес уже во время выполнения программы, занести результат вычислений в регистр процессора и воспользоваться косвенной адресацией.

Адрес, по которому очередная машинная команда произведёт обращение к памяти (неважно, задан ли этот адрес явно или вычислен) называется ***исполнительным адресом***. В предыдущем параграфе мы рассматривали ситуации, когда адрес вычислен, результат вычислений занесён в регистр и именно значение, хранящееся в регистре, используется в качестве исполнительного адреса. Для удобства программирова-

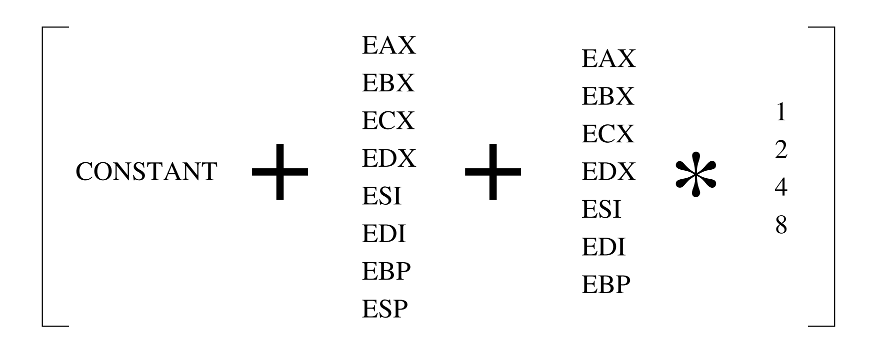


Рис. 2.2. Общий вид исполнительного адреса

ния процессор i386 позволяет и более сложные способы задания исполнительного адреса, при которых *исполнительный адрес вычисляется уже в ходе выполнения команды.*

Если говорить точнее, мы можем потребовать от процессора взять некоторое заранее заданное значение (возможно, равное нулю, а возможно, и не нулевое), прибавить к нему значение, хранящееся в одном из регистров, а затем взять значение, хранящееся в ещё одном из регистров, умножить на 1, 2, 4 или 8 и прибавить результат к уже имеющемуся адресу. Например, мы можем написать

```
        mov eax, [array+ebx+2*edi]
```

В результате такой команды процессор сложит число (заданное меткой `array`) с содержимым регистра `EBX` и удвоенным содержимым регистра `EDI`, результат такого сложения использует в качестве исполнительного адреса, извлечёт из области памяти по этому адресу 4 байта и скопирует их в регистр `EAX`. Каждое из трёх слагаемых, используемых в исполнительном адресе, является необязательным, то есть мы можем использовать только два слагаемых или всего одно (как, собственно, мы и поступали в предыдущих параграфах).

Важно понимать, что выражение в квадратных скобках никоим образом не может быть произвольным. Например, мы не можем взять три регистра, не можем умножить один регистр на 2, а другой на 4, не можем умножать на иные числа, кроме 1, 2, 4 и 8, не можем, например, перемножить два регистра между собой или вычесть значение регистра, вместо того чтобы прибавлять его. Общий вид исполнительного адреса показан на рис. 2.2; как можно заметить, в качестве регистра, подлежащего домножению на коэффициент, мы не можем использовать `ESP`, в качестве же регистра, значение которого просто добавляется к заданному адресу, можно использовать любой из восьми регистров общего назначения.

С другой стороны, ассемблер допускает определённые вольности с записью адреса, если только он при этом может корректно преобразовать адрес в машинную команду. Во-первых, слагаемые можно расположить в произвольном порядке. Во-вторых, можно использовать не одну константу, а две: ассемблер сам сложит их и результат запишет в получающуюся машинную команду. Наконец, можно умножить регистр на 3, 5 или 9: если вы напишете, например, [eax*5], ассемблер «переведёт» это как [eax+eax*4]. Конечно, если вы попытаетесь написать [eax+ebx*5], ассемблер выдаст ошибку, ведь нужное ему слагаемое вы уже использовали.

Чтобы понять, зачем может понадобиться такой сложный вид исполнительного адреса, достаточно представить себе *двумерный* массив, состоящий, например, из 10 строк, каждая из которых содержит 15 четырёхбайтных целых чисел. Назовём этот массив matrix, поставив перед его описанием соответствующую метку:

```
matrix  dd 10*15
```

Для доступа к элементам N-й строки такого массива мы можем вычислить смещение от начала массива до начала этой N-й строки (для этого нужно умножить N на длину строки, составляющую $15 * 4 = 60$ байт), занести результат вычислений, скажем, в EAX, затем в другой регистр (например, в EBX) занести номер нужного элемента в строке — и исполнительный адрес вида [matrix+eax+4*ebx] в точности задаст нам место в памяти, где расположен нужный элемент.

§ 2.2.7. Размеры операндов и их допустимые комбинации

Итак, мы ввели три типа операндов:

1. непосредственные операнды, задающее значение прямо в команде;
2. регистровые операнды, предписывающие взять значение из заданного регистра и/или поместить результат выполнения команды в этот регистр
3. операнды типа «память», задающие адрес, по которому в памяти находится нужное значение и/или по которому в память нужно записать результат работы команды.

Ясно, что не в любой ситуации нам подойдёт любой тип операнда. Например, очевидно, что непосредственный операнд нельзя использовать в качестве первого аргумента команды mov, ведь этот аргумент должен задавать то место, *куда* производится копирование данных; мы можем копировать данные в регистр или в область оперативной памяти, однако непосредственные операнды ни того, ни другого не задают. Имеются и

другие ограничения, налагаемые, как правило, устройством самого процессора как электронной схемы. Так, например, ни в команде `mov`, ни в других командах нельзя использовать сразу два операнда типа «память». Если необходимо, скажем, скопировать значение из области памяти `x` в область памяти `y`, необходимо делать это через регистр:

```
mov eax, [x]
mov [y], eax
```

Команда `mov [y],[x]` будет отвергнута ассемблером как ошибочная, поскольку ей не соответствует никакой машинный код: процессор попросту *не умеет* выполнять такое копирование за одну инструкцию.

Все остальные комбинации типов операндов для команды `mov` являются допустимыми, то есть за одну команду `mov` мы можем:

1. скопировать значение из регистра в регистр
2. скопировать значение из регистра в память
3. скопировать значение из памяти в регистр
4. задать (непосредственным операндом) значение регистра
5. задать (непосредственным операндом) значение ячейки или области памяти.

Последний вариант заслуживает особого рассмотрения. До сих пор во всех командах, которые мы использовали в примерах, хотя бы один из операндов был регистровым; это позволяло не думать о *размере* операндов, то есть о том, являются ли наши операнды отдельными байтами, двухбайтовыми «словами» или четырёхбайтовыми «двойными словами». Отметим, что команда `mov` не может пересылать данные между операндами разного размера (например, между однобайтовым регистром `AL` и двухбайтовым регистром `CX`); поэтому всегда, если хотя бы один из операндов является регистровым, можно однозначно сказать, какого размера порция данных подлежит обработке (в данном случае простому копированию). Однако же в варианте, когда первый операнд команды `mov` задаёт адрес в памяти, куда нужно записать значение, а второй является непосредственным (то есть записываемое значение задано прямо в команде), ассемблер не знает и не имеет оснований предполагать, какого конкретно размера нужно переслать порцию данных, или, иначе говоря, сколько байт памяти, начиная с заданного адреса, должно быть записано. Поэтому, например, команда

```
mov [x], 25    ; ОШИБКА!!!
```

будет отвергнута как ошибочная: непонятно, имеется ли в виду *байт* со значением 25, *«слово»* со значением 25 или *«двойное слово»* со значением 25. Тем не менее, команда, подобная вышеприведённой, вполне может понадобиться, и процессор умеет такую команду выполнять. Чтобы воспользоваться такой командой, нам нужно просто указать ассемблеру, что конкретно мы имеем в виду. Это делается указанием ***спецификатора размера*** перед любым из операндов; в качестве такого спецификатора может выступать слово `byte`, `word` или `dword`, обозначающие, соответственно, байт, слово или двойное слово (т. е. размер 1, 2 или 4 байта). Так, например, если мы хотели записать число 25 в четырёхбайтную область памяти, находящуюся по адресу `x`, мы можем написать

```
mov [x], dword 25
```

или

```
mov dword [x], 25
```

Сделаем одно важное замечание. Различные машинные команды, выполняющие схожие действия, могут обозначаться одной и той же мнемоникой. Так,

```
mov eax, 2
mov eax, [x]
mov [x], eax
mov [x], al
```

представляют собой четыре совершенно разные машинные команды, они имеют **разные значения машинного кода** и даже занимают разное количество байтов в памяти. Вместе с тем, команды

```
mov eax, 2
mov eax, x
```

используют один и тот же машинный код операции и различаются только значением второго операнда, который в обоих случаях непосредственный (действительно, ведь метка `x` будет заменена на адрес, то есть просто число).

§ 2.2.8. Команда `lea`

Возможности процессора по вычислению исполнительного адреса можно задействовать и отдельно от обращения к памяти. Для этого предусмотрена команда `lea` (название образовано от слов «load effective address»). Команда имеет два операнда, причём первый из них обязан быть регистровым (размером 2 или 4 байта), а второй — операндом типа

«память». При этом никакого обращения к памяти команда не делает; вместо этого в регистр, указанный первым операндом, заносится *адрес*, вычисленный обычным способом для второго операнда. Если первый операнд — двухбайтный регистр, то в него будут записаны младшие 16 бит вычисленного адреса. Например, команда

```
lea eax, [1000+ebx+8*ecx]
```

возьмёт значение регистра ECX, умножит его на 8, прибавит к этому значение регистра EBX и число 1000, а полученный результат занесёт в регистр EAX. Разумеется, вместо числа можно использовать и метку. Ограничения на выражение в скобках точно такие же, как и в других случаях использования операнда типа «память» (см. рис. 2.2 на стр. 49).

Подчеркнём ещё раз, что **команда lea только вычисляет адрес, не обращаясь к памяти**, несмотря на использование операнда типа «память».

§ 2.3. Целочисленная арифметика

§ 2.3.1. Простые команды сложения и вычитания

Операции сложения и вычитания над целыми числами производятся соответственно командами add и sub. Обе команды имеют по два операнда, причём первый из них задаёт и одно из чисел, участвующих в операции, и место, куда следует записать результат; второй операнд задаёт второе число для операции (второе слагаемое, либо вычитаемое). Ясно, что первый операнд обязан быть регистровым либо типа «память»; второй операнд у обеих команд может быть любого типа. Как и для команды mov, для команд add и sub нельзя использовать два операнда типа «память» одновременно.

Например, команда

```
add eax, ebx
```

означает «взять значение из регистра EAX, прибавить к нему значение из регистра EBX, а результат записать обратно в регистр EAX». Команда

```
sub [x], ecx
```

означает «взять четырёхбайтное число из памяти по адресу x, вычесть из него значение из регистра ECX, результат записать обратно в память по тому же адресу». Команда

```
add edx, 12
```

увеличит на 12 содержимое регистра EDX, а команда

```
        add dword [x], 12
```

сделает то же самое с четырёхбайтной областью памяти по адресу x; обратите внимание, что нам пришлось явно указать размер операнда (см. § 2.2.7, стр. 51).

Интересно, что команды add и sub работают правильно вне зависимости от того, считаем ли мы их операнды числами знаковыми или беззнаковыми[7]. В зависимости от полученного результата команды add и sub выставляют значения *флагов* OF, CF, ZF и SF (см. стр. 36), однако не всегда эти флаги имеет смысл рассматривать.

Флаг ZF устанавливается в единицу, если в результате последней операции получился ноль, в противном случае флаг сбрасывается; ясно, что значение этого флага осмысленно как для знаковых, так и для беззнаковых чисел.

Флаг SF устанавливается в единицу, если получено отрицательное число, иначе он сбрасывается в ноль. Процессор производит установку этого флага, попросту копируя в него старший бит результата; для знаковых чисел этот бит действительно означает знак числа, но для беззнаковых значение флага SF не имеет никакого смысла.

Флаг OF устанавливается, если произошло *переполнение*, что означает, что в результате сложения двух положительных получилось отрицательное, либо, наоборот, в результате сложения двух отрицательных получилось положительное, и т. д. Ясно, что этот флаг, как и предыдущий, не имеет никакого смысла для беззнаковых чисел.

Наконец, флаг CF устанавливается, если (для беззнаковых чисел) произошел перенос из старшего разряда, либо произошел заём из несуществующего разряда. По смыслу этот флаг является аналогом OF в применении к беззнаковым числам (результат не поместился в размер операнда, либо получился отрицательным). Для знаковых чисел этот флаг смысла не имеет.

Подчеркнём, что **при сложении и вычитании процессор не знает, работает ли он со знаковыми или с беззнаковыми числами**. Схематически сложение и вычитание производится абсолютно одинаково вне зависимости от «знаковости» операндов; флаги процессор выставляет все, т. е. и те, что имеют смысл только для знаковых, и те, что имеют смысл только для беззнаковых. Помнить о том, какие числа имеются в виду — это обязанность программиста; именно программист должен использовать набор флагов, соответствующий знаковости обрабатываемых чисел.

---

[7]Знаковость и беззнаковость целых чисел мы обсуждали в §§ 1.3.1 и 1.3.2; если вы не чувствуете уверенности в обращении с этими терминами, обязательно перечитайте эти параграфы и при необходимости задайте вопросы преподавателю, в противном случае вы рискуете ничего не понять в дальнейшем курсе.

§ 2.3.2. Сложение и вычитание с переносом

Наличие флага переноса позволяет организовать сложение и вычитание чисел, не помещающихся в регистры, способом, напоминающим школьное сложение и вычитание «в столбик». Для этого в процессоре i386 предусмотрены команды `adc` и `sbb`. По своей работе и свойствам они полностью аналогичны командам `add` и `sub`, но отличаются от них тем, что учитывают значение флага переноса (`CF`) на момент начала выполнения операции. Команда `adc` *добавляет* к своему итоговому результату значение флага переноса, команда `sbb`, напротив, *вычитает* значение флага переноса из своего результата. После того как результат сформирован, обе команды заново выставляют все флаги, включая и `CF`, уже в соответствии с новым результатом.

Приведём пример. Пусть у нас есть два 64-битных целых числа, причём первое записано в регистры `EDX` (старшие 32 бита) и `EAX` (младшие 32 бита), а второе точно так же записано в регистры `EBX` и `ECX`. Тогда сложить эти два числа можно командами

```
add eax, ecx   ; складываем младшие части
adc edx, ebx   ; теперь старшие, с учётом переноса
```

если же нам понадобится произвести вычитание, то это делается командами

```
sub eax, ecx   ; вычитаем младшие части
sbb edx, ebx   ; теперь старшие, с учётом заёма
```

§ 2.3.3. Команды `inc`, `dec`, `neg` и `cmp`

Чтобы завершить рассмотрение простейших арифметических операций, опишем ещё четыре команды.

Команды `inc` и `dec`, с которыми мы уже сталкивались в ранее приведённых примерах имеют всего один операнд (регистровый или типа «память») и производят, соответственно, увеличение и уменьшение на единицу. Обе команды устанавливают флаги `ZF`, `OF` и `SF`, но не затрагивают флаг `CF`. Отметим, что при использовании этих команд с операндом типа «память» указание размера операнда оказывается *обязательным*: действительно, для ассемблера нет другого способа понять, какого размера область памяти имеется в виду.

Команда `neg`, также имеющая один операнд, обозначает *смену знака*, то есть операцию «унарный минус». Обычно её применяют к знаковым числам; тем не менее, она устанавливает все четыре флага `ZF`, `OF` и `SF` и `CF`, как если бы операнд вычитался из нуля.

Наконец, команда `cmp` (от слова «compare» — «сравнить») производит точно такое же вычитание, как и команда `sub`, за исключением того, что

результат никуда не записывается. Команда вызывается ради установки флагов, обычно сразу после неё следует команда условного перехода.

§ 2.3.4. Целочисленное умножение и деление

В отличие от сложения и вычитания, умножение и деление схематически реализуется сравнительно сложно[8], так что команды умножения и деления могут показаться организованными очень неудобно для программиста. Причина этого, по-видимому, в том, что создатели процессора i386 и его предшественников действовали здесь прежде всего из соображений удобства реализации самого процессора.

Надо сказать, что умножение и деление доставляет некоторые сложности не только разработчикам процессоров, но и программистам, и отнюдь не только в силу неудобности соответствующих команд, но и по самой своей природе. Во-первых, в отличие от сложения и вычитания, умножение и деление для знаковых и беззнаковых чисел производится совершенно по-разному, так что небходимы и различные команды.

Во-вторых, интересные вещи происходят с размерами операндов. При умножении размер (количество значащих битов) результата может быть *вдвое* больше, чем размер исходных операндов, так что, если мы не хотим потерять информацию, то одним флажком, как при сложении и вычитании, мы тут не обойдёмся: нужен дополнительный регистр для хранения старших битов результата. С делением ситуация ещё интереснее: если модуль делителя превосходит `1`, размер результата будет меньше размера делимого (если точнее, *количество значащих битов* результата двоичного деления не превосходит $n-m+1$, где n и m — количество значащих битов делимого и делителя соответственно), так что желательно иметь возможность задавать делимое более длинное, чем делитель и результат. Кроме того, целочисленное деление даёт в качестве результата не одно, а два числа: частное и остаток. Разделять между собой операции нахождения частного и остатка нежелательно, поскольку может привести к двухкратному выполнению (на уровне электронных схем) одних и тех же действий.

Все команды целочисленного умножения и деления имеют только один операнд[9], задающий второй множитель в командах умножения и делитель в командах деления, причём этот операнд может быть регистровым или типа «память», но не непосредственным. Что касается первого множителя и делимого, то для их задания, а также в качестве

[8] На некоторых процессорах, даже современных, этих операций вообще нет, и причина этого — исключительно сложность их реализации.

[9] На самом деле, из этого правила есть исключение: команда целочисленного умножения знаковых чисел `imul` имеет двухместную и даже трёхместную формы, но рассматривать эти формы мы не будем: пользоваться ими ещё сложнее, чем обычной одноместной формой.

разрядн. (бит)	умножение		деление		
	неявный множитель	результат умножения	делимое	частное	остаток
8	AL	AX	AX	AL	AH
16	AX	DX:AX	DX:AX	AX	DX
32	EAX	EDX:EAX	EDX:EAX	EAX	EDX

Таблица 2.1. Расположение неявного операнда и результатов для операций целочисленного деления и умножения в зависимости от разрядности явного операнда

цели для записи результата используются ***неявный операнд***, в качестве которого в данном случае выступают регистры AL, AX, EAX, а при необходимости — и регистровые пары DX:AX и EDX:EAX (напомним, что буква A означает слово «аккумулятор»; это и есть особая роль регистра EAX, о которой говорилось на стр. 34).

Для умножения беззнаковых чисел применяют команду mul, для умножения знаковых — команду imul. В обоих случаях, в зависимости от разрядности операнда (второго множителя) первый множитель берётся из регистра AL (для однобайтной операции), либо AX (для двухбайтной операции), либо EAX (для четырёхбайтной), а результат помещается в регистр AX (если операнды были однобайтными), либо в регистровую пару DX:AX (для двухбайтной операции), либо в регистровую пару EDX:EAX (для четырёхбайтной операции). Это можно более наглядно представить в виде таблицы (см. табл. 2.1).

Команды mul и imul устанавливают флаги CF и OF в ноль, если старшая половина результата равна нулю (то есть все значащие биты результата уместились в младшей половине), в противном случае CF и OF устанавливаются в единицу. Значения остальных флагов после выполнения mul и imul *не определены* (то есть ничего осмысленного сказать об их значениях нельзя, причём разные процессоры могут устанавливать их по-разному и даже в результате выполнения одной и той же команды на одном и том же процессоре флаги могут получить разные значения).

Для деления (и нахождения остатка от деления) целых чисел применяют команду div (для беззнаковых) и idiv (для знаковых). Единственный операнд команды, как уже говорилось выше, задаёт *делитель*. В зависимости от разрядности этого делителя (1, 2 или 4 байта) делимое берётся из регистра AX, регистровой пары DX:AX или регистровой пары EDX:EAX, частное помещается в регистр AL, AX или EAX, а остаток от деления — в регистры AH, DX или EDX, соответственно (см. табл. 2.1). Частное всегда округляется в сторону нуля (для беззнаковых и положительных — в меньшую, для отрицательных — в бо́льшую сторону). Знак

остатка, вычисляемого командой `imul`, всегда совпадает со знаком делимого, а абсолютная величина (модуль) остатка всегда строго меньше модуля делителя. Значения флагов после выполнения целочисленного деления не определены.

Отдельного рассмотрения заслуживает ситуация, когда в делителе на момент выполнения команды `div` или `idiv` находится число 0. Делить на ноль, как известно, нельзя, а собственных средств, чтобы сообщить о происшедшей ошибке, у процессора нет. Поэтому процессор инициирует так называемое *внутреннее прерывание*, в результате которого управление получает операционная система; в большинстве случаев она сообщает об ошибке и завершает текущую задачу как аварийную. То же самое произойдёт и в случае, если результат деления не уместился в отведённые ему разряды: например, если мы занесём в `EDX` число `10h`, а в `EAX` — любое другое, даже просто 0, и попытаемся поделить это (то есть шестнадцатеричное 1000000000, или 2^{36}), скажем, на 2 (записав его, например, в `EBX`, чтобы сделать деление 32-разрядным), то результат (2^{35}) в 32 разряда «не влезет», и процессору придётся инициировать прерывание. Подробнее о прерываниях мы расскажем в § 4.2.

§ 2.4. Условные и безусловные переходы

Как уже отмечалось, в обычное последовательное выполнение команд можно вмешаться, выполнив ***передачу управления***, называемую также ***переходом***. Различают команды ***безусловных переходов***, выполняющие передачу управления в другое место программы без всяких проверок, и команды ***условных переходов***, которые могут, в зависимости от результата проверки некоторого условия, либо выполнить переход в заданную точку, либо не выполнять его — в этом случае выполнение программы, как обычно, продолжится со следующей команды.

§ 2.4.1. Безусловный переход и виды переходов

В системе команд процессора i386 все команды передачи управления подразделяются, в зависимости от «дальности» такой передачи, на *три типа*.

1. ***Дальние*** (`far`) переходы подразумевают передачу управления во фрагмент программы, расположенный *в другом сегменте*. Поскольку под управлением ОС Unix мы используем «плоскую» модель памяти, в которой разделение на сегменты отсутствует (точнее, имеет место лишь один сегмент, «накрывающий» всё наше виртуальное адресное пространство), такие переходы нам понадобиться не могут: у нас попросту нет других сегментов.

2. ***Близкие*** (`near`) переходы — это передача управления в произвольное место внутри одного сегмента; фактически такие переходы

представляют собой явное изменение значения EIP. В «плоской» модели памяти это именно тот вид переходов, с помощью которого мы можем «прыгнуть» в произвольное место в нашем адресном пространстве.

3. ***Короткие*** (short) переходы используются для оптимизации в случае, если точка, куда надлежит «прыгнуть», отстоит от текущей команды не более чем на 127 байт вперёд или 128 байт назад. В машинном коде такой команды смещение задаётся всего одним байтом, отсюда соответствующее ограничение.

При написании команды перехода мы можем явно указать вид нужного нам перехода, поставив после команды слово short или near (ассемблер понимает, разумеется, и слово far, но нам это не нужно). Если этого не сделать, ассемблер выбирает тип перехода *по умолчанию*, причём для безусловных переходов это near, что нас обычно устраивает, а вот для условных переходов по умолчанию используется short. Вытекающие из этого сложности и способы их преодоления мы обсудим в следующем параграфе, который посвящён условным переходам, а пока вернёмся к переходам безусловным.

Команда безусловного перехода называется jmp (от слова «jump», которое буквально переводится как «прыжок»). У команды предусмотрен один операнд, определяющий собственно адрес, куда следует передать управление. Чаще всего используется форма команды jmp с непосредственным операндом, то есть адресом, указанным прямо в команде. Естественно, указываем мы не числовой адрес (которого обычно не знаем), а метку. Возможно, однако, использовать и регистровый операнд (в этом случае переход производится по адресу, взятому из регистра), и операнд типа «память» (адрес читается из двойного слова, расположенного в заданной позиции в памяти); такие переходы называют ***косвенными***, в отличие от ***прямых***, для которых адрес задаётся явно. Приведём несколько примеров:

```
jmp cycle   ; переход на метку cycle
jmp eax     ; переход по адресу из регистра EAX
jmp [addr]  ; переход по адресу, содержащемуся
            ; в памяти, которая помечена меткой addr
jmp [eax]   ; переход по адресу, прочитанному из
            ; памяти, расположенной по адресу,
            ; взятому из регистра EAX
```

Здесь первая команда задаёт прямой переход, а остальные — косвенный.

Если метка, на которую нужно перейти, находится достаточно близко к текущей позиции, можно попытаться соптимизировать машинный код, применив слово short:

```
mylabel:
        ; ...
        ; небольшое количество команд
        ; ...
        jmp short mylabel
```

На глаз обычно тяжело определить, действительно ли метка находится достаточно близко, тем более что макросы (например, GETCHAR) могут сгенерировать целый ряд команд, иногда слабо предсказуемый по длине. Но на этот счёт можно не беспокоиться: если расстояние до метки окажется больше допустимого, ассемблер выдаст ошибку примерно такого вида:

```
file.asm:35: error: short jump is out of range
```

и останется только найти строку с указанным номером (в данном случае 35) и убрать «несработавшее» слово short.

§ 2.4.2. Условные переходы по отдельным флагам

В противоположность командам безусловного перехода, команды условного перехода ассемблер по умолчанию считает «короткими», если не указать тип перехода явно.

Такой, на первый взгляд, странный подход к командам переходов обусловлен историческими причинами: на ранних процессорах линейки x86 условные переходы были только короткими, других команд просто не было. Процессор i386 и все более поздние процессоры, конечно же, поддерживают и близкие условные переходы; дальние условные переходы до сих пор не поддерживаются, но нам они всё равно не нужны.

Простейшие команды условного перехода производят переход по указанному адресу в случае, если один из флагов равен нулю (сброшен) или единице (установлен). Имена этих команд образуются из буквы J (от слова «jump», первой буквы названия флага (например, Z для флага ZF) и, возможно, вставленной между ними буквы N (от слова «not»), если переход нужно произвести при условии равенства флага нулю. Все эти команды приведены в табл. 2.2. Напомним, что смысл каждого из флагов мы рассмотрели на стр. 36.

Такие команды условного перехода обычно ставят непосредственно после арифметической операции (например, сразу после команды cmp, см. стр. 55). Например, две команды

```
cmp eax, ebx
jz are_equal
```

можно прочитать как приказ «сравнить значения в регистрах EAX и EBX и если они равны, перейти на метку are_equal».

команда	условие перехода	команда	условие перехода
jz	ZF=1	jnz	ZF=0
js	SF=1	jns	SF=0
jc	CF=1	jnc	CF=0
jo	OF=1	jno	OF=0
jp	PF=1	jnp	PF=0

Таблица 2.2. Простейшие команды условного перехода

§ 2.4.3. Переходы по результатам сравнений

Если нам нужно сравнить два числа на *равенство*, всё довольно просто: достаточно, как в предыдущем примере, воспользоваться флагом ZF. Но что делать, если нас интересует, например, условие $a < b$? Сначала мы, естественно, применим команду

cmp a, b

(в качестве a и b могут быть любые операнды, нужно только помнить, что они не могут быть оба одновременно операндами типа «память»). Команда выполнит сравнение своих операндов — точнее говоря, вычтет из a значение b и соответствующим образом выставит значения флагов. Но вот дальнейшее, как мы сейчас увидим, оказывается несколько сложнее.

Если числа a и b — знаковые, то на первый взгляд всё просто: вычитание $a - b$ при условии $a < b$ даёт число строго отрицательное, так что флаг знака (SF, sign flag) должен быть установлен, и мы можем воспользоваться командой js или jns. Но ведь результат мог и не поместиться в длину операнда (например, в 32 бита, если мы сравниваем 32-разрядные числа), то есть могло возникнуть переполнение! В этом случае значение флага SF окажется прямо противоположным ожидавшемуся, зато будет взведён флаг OF (overflow flag). Таким образом, условие $a < b$ выполняется в двух случаях: если SF=1, но OF=0 (то есть переполнения не было, число получилось отрицательное), либо если SF=0, но OF=1 (число получилось положительное, но это результат переполнения, а на самом деле результат отрицательный). Иначе говоря, нас интересует, чтобы флаги SF и OF *не были равны друг другу*: SF≠OF. Для такого случая в процессоре i386 предусмотрена команда jl (от слов «**j**ump if **l**ess than»), обозначаемая также мнемоникой jnge («jump if not greater or equal»).

Рассмотрим теперь ситуацию, если числа a и b — беззнаковые. Как мы уже обсуждали в § 2.3.1 (см. стр. 54), по итогам арифметических операций над беззнаковыми числами флаги OF и SF рассматривать не имеет

имя ком.	jump if...	выр. $a \vee b$	условие перехода	синоним
равенство				
`je` `jne`	equal not equal	$a = b$ $a \neq b$	`ZF`$= 1$ `ZF`$= 0$	`jz` `jnz`
неравенства для знаковых чисел				
`jl` `jnge`	less not greater or equal	$a < b$	`SF`$\neq$`OF`	
`jle` `jng`	less or equal not greater	$a \leqslant b$	`SF`$\neq$`OF` или `ZF`$= 1$	
`jg` `jnle`	greater not less or equal	$a > b$	`SF`$=$`OF` и `ZF`$= 0$	
`jge` `jnl`	greater or equal not less	$a \geqslant b$	`SF`$=$`OF`	
неравенства для беззнаковых чисел				
`jb` `jnae`	below not above or equal	$a < b$	`CF`$= 1$	`jc`
`jbe` `jna`	below or equal not above	$a \leqslant b$	`CF`$= 1$ или `ZF`$= 1$	
`ja` `jnbe`	above not below or equal	$a > b$	`CF`$= 0$ и `ZF`$= 0$	
`jae` `jnb`	above or equal not below	$a \geqslant b$	`CF`$= 0$	`jnc`

Таблица 2.3. Команды условного перехода по результатам арифметического сравнения (`cmp` a, b)

смысла, но зато осмысленным становится рассмотрение флага `CF` (carry flag), который выставляется в единицу, если по итогам арифметической операции произошел перенос из старшего разряда (при сложении), либо заём из несуществующего разряда (для вычитания). Именно это нам здесь и нужно: если a и b рассматриваются как беззнаковые и $a < b$, то при вычитании $a - b$ как раз и произойдёт такой заём. Таким образом, нам достаточно воспользоваться значением флага `CF`, то есть выполнить команду `jc`, которая специально для данной ситуации имеет синонимы `jb` («jump if below») и `jnae` («jump if not above or equal»).

Когда нас интересуют соотношения «больше» и «меньше либо равно», необходимо включить в рассмотрение и флаг `ZF`, который (как для знаковых, так и для беззнаковых чисел) обозначает равенство аргументов предшествующей команды `cmp`.

Все команды условных переходов по результату арифметического сравнения приведены в табл. 2.3.

§ 2.4.4. Условные переходы и регистр ECX; циклы

Как уже говорилось, некоторые регистры общего назначения в некоторых случаях имеют особую роль; в частности, регистр ECX лучше других приспособлен к роли *счётчика цикла*. Выражается это в том, что в системе команд процессора i386 имеются специальные команды, учитывающие значение ECX, а для других регистров таких команд нет.

Одна из таких команд называется loop и предназначена для организации циклов с заранее известным количеством итераций. В качестве счётчика цикла она использует регистр ECX, в который перед началом цикла необходимо занести нужное число итераций. Сама команда loop выполняет два действия: уменьшает на единицу значение в регистре ECX и, если в результате значение не стало равным нулю, производит переход на заданную метку.

Отметим, что команда loop имеет одно важное ограничение: она выполняет только «короткие» переходы, то есть с её помощью невозможно осуществить переход на метку, отстоящую от самой команды более чем на 128 байт.

Пусть, например, у нас есть массив из 1000 двойных слов, заданный с помощью директивы

```
array   resd 1000
```

и мы хотим посчитать сумму его элементов. Это можно сделать с помощью следующего фрагмента кода:

```
        mov ecx, 1000    ; кол-во итераций
        mov esi, array   ; адрес первого элемента
        mov eax, 0       ; начальное значение суммы
lp:     add eax, [esi]   ; прибавляем число к сумме
        add esi, 4       ; адрес следующего элемента
        loop lp          ; уменьшаем счётчик;
                         ; если нужно - продолжаем
```

Здесь мы использовали фактически две переменные цикла — регистр ECX в качестве счётчика и регистр ESI для хранения адреса текущего элемента массива.

Конечно, можно произвести аналогичное действие и для любого другого регистра общего назначения, воспользовавшись двумя командами. Например, мы можем уменьшить на единицу регистр EAX и осуществить переход на метку lp при условии, что полученный в EAX результат не равен нулю; это будет выглядеть так:

```
        dec eax
        jnz lp
```

Точно так же можно записать две команды и для регистра ECX:

```
dec ecx
jnz lp
```

Однако команда `loop lp`, делая те же действия, работает быстрее и занимает меньше памяти.

В примере с массивом можно обойтись и без ESI, одним только счётчиком:

```
        mov ecx, 1000
        mov eax, 0
lp:     add eax, [array+4*ecx-4]
        loop lp
```

Здесь есть два интересных момента. Во-первых, массив мы вынуждены проходить с конца в начало. Во-вторых, исполнительный адрес в команде `add` имеет несколько странный вид. Действительно, регистр ECX пробегает значения от 1000 до 1 (для нулевого значения цикл уже не выполняется), тогда как адреса элементов массива пробегают значения от `array+4*999` до `array+4*0`, так что умножать на 4 следовало бы не ECX, а (`ecx-1`). Однако этого мы сделать не можем и просто вычитаем 4. На первый взгляд это противоречит сказанному в § 2.2.6 относительно общего вида исполнительного адреса (слагаемое в виде константы должно быть одно, либо ни одного), однако на самом деле ассемблер NASM прямо во время трансляции вычтет значение 4 из значения `array` и уже в таком виде оттранслирует, так что в итоговом машинном коде константное слагаемое как раз и будет одно.

Рассмотрим теперь две дополнительные команды условного перехода. Команда `jcxz` (jump if CX iz zero) производит условный переход, *если в регистре CX содержится ноль*. Флаги при этом не учитываются. Аналогичным образом команда `jecxz` производит переход, если ноль содержится в регистре ECX. Как и для команды `loop`, этот переход всегда короткий. Чтобы понять, зачем введены эти команды, представьте себе, что на момент входа в цикл в регистре ECX *уже* содержится ноль. Тогда сначала выполнится тело цикла, а потом команда `loop` уменьшит счётчик на единицу, в результате чего счётчик окажется равен максимально возможному целому беззнаковому числу (двоичная запись этого числа состоит из всех единиц), так что тело цикла будет выполнено 2^{32} раз, тогда как по смыслу его, скорее всего, не следовало выполнять вообще. Чтобы избежать таких неприятностей, перед циклом можно поставить команду `jecxz`:

```
        ; заполняем ecx
        jecxz lpq
lp:     ; тело цикла
        ; ...
        loop lp
lpq:
```

В заключение рассмотрим две модификации команды `loop`. Команда `loope`, называемая также `loopz`, производит переход, если в регистре `ECX` — не ноль и при этом флаг `ZF` установлен, тогда как команда `loopne` (или, что то же самое, `loopnz`) — если в регистре `ECX` не ноль и флаг `ZF` сброшен.

§ 2.5. Побитовые операции

§ 2.5.1. Логические операции

Информацию, записанную в регистры и память в виде байтов, слов и двойных слов можно рассматривать не только как представление целых чисел, но и как строки, состоящие из отдельных и (в общем случае) никак не связанных между собой битов.

Для работы с такими битовыми строками используются специальные команды ***побитовых операций***. Простейшими из них являются двухместные команды `and`, `or` и `xor`, выполняющие соответствующую логическую операцию («и», «или», «исключающее или») отдельно над первыми битами обоих операндов, отдельно над вторыми битами и т. д.; результат, представляющий собой битовую строку той же длины, что и операнды, заносится, как обычно для арифметических команд, в регистр или область памяти, определяемую первым операндом. Ограничения на используемые операнды у этих команд такие же, как и у двухместных арифметических команд: первый операнд должен быть либо регистровым, либо типа «память», второй операнд может быть любого типа; нельзя использовать операнд типа «память» одновременно для первого и второго операнда; если ни один из операндов не является регистровым, необходимо указать разрядность операции с помощью одного из слов `byte`, `word` и `dword`. Осуществить побитовое отрицание (инверсию) можно с помощью команды `not`, имеющей один операнд. Операнд может быть регистровый или типа «память»; в последнем случае, естественно, необходимо задать длину операнда словом `byte`, `word` или `dword`. Все эти команды устанавливают флаги `ZF`, `SF` и `PF` в соответствии с результатом; обычно используется только флаг `ZF`.

В программах на языке ассемблера очень часто встречается команда `xor`, оба операнда которой представляют собой один и тот же регистр, например,

```
        xor eax, eax
```

Это означает *обнуление* указанного регистра, т. е. то же самое, что и

```
        mov eax, 0
```

Команду `xor` для этого используют, потому что она занимает меньше места (2 байта против 5 для команды `mov`) и работает на несколько тактов быстрее. Некоторые программисты вместо `mov eax,-1` предпочитают использовать две команды

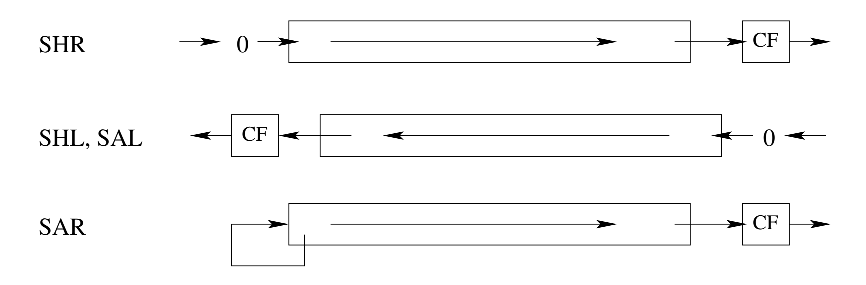


Рис. 2.3. Схема работы команд побитового сдвига

`xor eax,eax` и `not eax`, хотя выигрыш тут уже не столь заметен (4 байта кода против 5), а по времени исполнения тут можно и проиграть.

В случае, если необходимо просто проверить наличие в числе одного из заданных битов, может оказаться удобной команда `test`, которая работает так же, как и команда `and` (то есть выполняет побитовое «и» над своими операндами), но результат никуда не записывает, а только выставляет флаги.

В частности, для проверки на равенство нулю вместо

```
cmp eax, 0
```

часто используют команду

```
test eax, eax
```

которая занимает меньше памяти и работает быстрее.

§ 2.5.2. Операции сдвига

Часто приходится применять ***операции побитового сдвига***. Простейшие из них — команды ***простого побитового сдвига*** `shr` (**sh**ift **r**ight) и `shl` (**sh**ift **l**eft). Команды имеют два операнда, первый из который указывает, *что* сдвигать, а второй — на сколько битов производить сдвиг. Первый операнд может быть регистровым или типа «память» (во втором случае обязательно указание разрядности). Второй операнд может быть либо непосредственным, то есть числом от 1 до 31 (на самом деле, можно указать любое число, но от него будут использоваться только младшие пять разрядов), либо *регистром CL*; никакие другие регистры использовать нельзя. При выполнении этих команд с регистром `CL` в качестве второго операнда процессор игнорирует все разряды `CL`, кроме пяти младших.

Схема сдвига на 1 бит следующая. При сдвиге влево старший бит сдвигаемого числа переносится во флаг `CF`, остальные биты сдвигаются

влево (то есть бит с номером[10] n получает значение, которое до операции имел бит с номером $n-1$), в младший бит записывается ноль. При сдвиге вправо, наоборот, во флаг `CF` заносится младший бит, все биты сдвигаются вправо (то есть бит с номером n получает значение, которое до операции имел бит с номером $n+1$), в старший бит записывается ноль.

Отметим, что для **беззнаковых** чисел сдвиг на n бит влево эквивалентен умножению на 2^n, а сдвиг вправо — целочисленному делению на 2^n с отбрасыванием остатка. Интересно, что для **знаковых** чисел ситуация со сдвигом влево абсолютно аналогична, а вот сдвиг вправо для любого отрицательного числа даст положительное, ведь в знаковый бит будет записан ноль. Поэтому наряду с командами простого сдвига вводятся также и команды ***арифметического побитового сдвига*** `sal` (**s**hift **a**rithmetic **l**eft) и `sar` (**s**hift **a**rithmetic **r**ight). Команда `sal` делает то же самое, что и команда `shl` (на самом деле, это одна и та же машинная команда). Что касается команды `sar`, то она работает аналогично команде `shr`, за исключением того, что в старшем бите значение сохраняется таким же, каким оно было до операции; таким образом, если рассматривать сдвигаемую битовую строку как запись знакового целого числа, то операция `sar` не изменит знак числа (положительное останется положительным, отрицательное — отрицательным). Иначе говоря, операция арифметического сдвига вправо эквивалентна делению на 2^n с отбрасыванием остатка *для знаковых целых чисел*. Операции простых и арифметических сдвигов схематически показаны на рис. 2.3.

Команды побитовых сдвигов работают гораздо быстрее, чем команды умножения и деления; кроме того, обращаться с ними существенно легче: можно использовать любые регистры, так что не нужно думать о высвобождении аккумулятора. Поэтому при умножении и делении на степени двойки программисты практически всегда используют именно команды побитовых сдвигов. Более того, компиляторы языков высокого уровня при трансляции арифметических выражений тоже, как правило, стараются использовать сдвиги вместо умножений и делений, если это возможно.

Кроме рассмотренных, процессор i386 поддерживает также команды «сложных» побитовых сдвигов `shrd` и `shld`, работающих через два регистра; команды ***циклического побитового сдвига*** `ror` и `rol`; команды циклического сдвига через флаг `CF` — `rcr` и `rcl`. Все эти команды мы рассматривать не будем; при желании читатель может освоить их самостоятельно.

[10] По традиции мы предполагаем, что биты занумерованы справа налево, начиная с нуля, то есть, например, в 32-битном числе младший бит имеет номер 0, а старший — номер 31.

§ 2.5.3. Пример

Одним из применений битовых строк в программировании является представление подмножеств из конечного числа исходных элементов; попросту говоря, у нас имеется конечное множество объектов (например, сотрудники какого-нибудь предприятия, или тумблеры на каком-нибудь пульте управления, или даже просто числа от `0` до `N`) и нам в программе нужна возможность представлять *подмножество* этого множества: какие из сотрудников в настоящее время находятся на работе; какие из тумблеров на пульте установлены в положение «включено»; какие из `N` спортсменов, участвующих в марафоне, прошли очередной контрольный пункт; и т. п. Наиболее очевидное представление для подмножества множества `N` элементов — это область памяти, содержащая `N` двоичных разрядов (так, если в множество могут входить числа от 0 до 511, нам потребуется 512 разрядов, то есть 64 однобайтовых ячейки), где каждому из `N` возможных элементов приписывается один разряд, и этот разряд будет равен единице, если соответствующий элемент входит в подмножество, и нулю в противном случае. Говорят, что каждому из `N` объектов присвоен один из двух *статусов*: либо «входит в множество» (`1`), либо «не входит в множество» (`0`).

Итак, пусть нам потребовалось подмножество множества из 512 элементов; это могут быть совершенно произвольные объекты, нас интересует только то, что у каждого из них есть уникальный номер — число от 0 до 511. Чтобы хранить такое множество, мы опишем массив из 16 «двойных слов» (напомним, что «двойное слово» содержит 32 бита и может, соответственно, хранить статус 32 разных объектов). Как обычно, элементы массива будем считать занумерованными (или имеющими *индексы*) от 0 до 15. Элемент массива с индексом 0 будет хранить статус объектов с номерами от 0 до 31, элемент с индексом 1 — статус объектов с номерами от 32 до 63, и т. д. При этом внутри самого элемента биты будем считать занумерованными справа налево, то есть самый младший разряд будет иметь номер 0, самый старший — номер 31. Например, статус объекта с номером 17 будет храниться в 17-м бите нулевого элемента массива; статус объекта с номером 37 — в 5-м бите первого элемента; статус объекта с номером 510 — в 29-м бите 15-го элемента массива. Вообще, чтобы по номеру объекта `X` узнать, в каком бите какого элемента массива хранится его статус, достаточно разделить `X` на 32 (количество бит в каждом элементе) с остатком. Частное будет соответствовать номеру элемента в массиве, остаток — номеру бита в этом элементе. Это можно было бы сделать с помощью команды `div`, но лучше будет вспомнить, что число 32 есть степень двойки (2^5), так что если взять младшие пять бит числа `X`, мы получим остаток от его деления на 32, а если выполнить для него побитовый сдвиг вправо на 5 позиций — результат будет равен

искомому частному. Например, пусть число X занесено в регистр EBX, и нам необходимо узнать соответствующий номер элемента и номер бита в элементе. Оба номера не превосходят 255 (точнее, номер элемента не превосходит 15, а номер бита не превосходит 32), так что результат мы можем разместить в однобайтовых регистрах; пусть это будут BL (для номера бита) и BH (для номера элемента массива). Поскольку занесение любых новых значений в BL и BH испортит содержимое регистра EBX как целого, логично будет сначала скопировать число куда-то ещё, например в EDX, потом в EBX обнулить все биты, кроме пяти младших (при этом и значение EBX как целого, и значение его младшего байта — регистра BL станут равны искомому остатку от деления; потом в EDX мы выполним сдвиг вправо и реультат (который полностью уместится в младшем байте регистра EDX, то есть в регистра DL) скопируем в BH:

```
mov     edx, ebx
and     ebx, 11111b  ; взяли 5 младших разрядов
shr     edx, 5       ; разделили остальное на 32
mov     bh, dl
```

Однако то же самое можно сделать и короче, без использования дополнительных регистров, ведь все нужные биты у нас с самого начала находятся в EBX. Младшие пять разрядов числа X — это нужный нам остаток от деления, а нужное нам частное — это *следующие* несколько (в данном случае — не более четырёх) разрядов. Когда в EBX занесли число X, эти разряды оказались в позициях, начиная с пятой, а нам нужно, чтобы они оказались в регистре BH, который есть ни что иное как второй байт регистра EBX, так что достаточно сдвинуть всё содержимое EBX *влево* на три позиции, и нужный нам результат деления аккуратно «впишется» в BH; после этого содержимое BL мы сдвинем обратно на те же три бита, что заодно и очистит нам его старшие биты:

```
shl     ebx, 3
shr     bl, 3
```

Научившись преобразовывать номер объекта в номер элемента массива и номер разряда в элементе, вернёмся к исходной задаче. Для начала опишем массив:

```
section .bss
set512  resd    16
```

Теперь у нас есть подходящая область памяти, и с адресом её начала связана метка set512. Где-то в начале программы (а возможно, и не только в начале) нам, видимо, понадобится операция очистки множества, то есть такой набор команд, после которого статус всех элементов оказывается

нулевой (в множество не входит ни один элемент). Для этого достаточно занести нули во все элементы массива, например, так:

```
section .text

; ...

        xor     eax, eax        ; eax := 0
        mov     ecx, 15
        mov     esi, set512
lp:     mov     [esi+4*ecx], eax
        loop    lp
```

Пусть теперь у нас в регистре EBX имеется номер элемента X, и нам необходимо внести элемент в множество, то есть установить соответствующий бит в единицу. Для этого мы сначала найдём номер бита в элементе массива и вычислим *маску* — такое число, в котором только один бит (как раз нужный нам) равен единице, а в остальных разрядах нули. Затем мы найдём нужный элемент массива и применим к нему и к маске операцию «или», результат которой занесём обратно в элемент массива. При этом нужный нам бит в элементе окажется равен единице, а остальные не изменятся. Для вычисления маски мы возьмём единицу и сдвинем её на нужное количество разрядов влево. Напомним, что из регистров только CL может быть вторым аргументом команд побитовых сдвигов, так что номер бита имеет смысл сразу вычислять в CL. Итак, пишем:

```
; внести в множество set512 элемент,
; номер которого находится в EBX
        mov     cl, bl              ; получаем номер бита
        and     cl, 11111b          ;   в регистре CL
        mov     eax, 1              ; создаём маску
        shl     eax, cl             ;   в регистре EAX
        mov     edx, ebx            ; вычисляем номер эл-та
        shr     edx, 5              ;   в регистре edx
        or      [set512+4*edx], eax ; применяем маску
```

Аналогично решается и задача по исключению элемента из множества, только маска на этот раз будет инвертирована (0 в нужном разряде, единицы во всех остальных), а применять мы её будем с командой and (логическое «и»), в результате чего нужный бит обнулится, остальные не изменятся:

```
; убрать из множества set512 элемент,
; номер которого находится в EBX
```

```
        mov     cl, bl              ; получаем номер бита
        and     cl, 11111b          ;   в регистре CL
        mov     eax, 1              ; создаём маску
        shl     eax, cl             ;   в регистре EAX
        not     eax                 ; инвертируем маску
        mov     edx, ebx            ; вычисляем номер эл-та
        shr     edx, 5              ;   в регистре edx
        and     [set512+4*edx], eax ; применяем маску
```

Узнать, входит ли элемент с заданным номером в множество, можно тоже с помощью маски (единица в нужном разряде, нули в остальных) и команды `test`. Результат покажет флаг `ZF`: если он будет взведён — значит, соответствующего элемента в множестве не было, и наоборот:

```
; узнать, входит ли в множество set512 элемент,
; номер которого находится в EBX
        mov     cl, bl              ; получаем номер бита
        and     cl, 11111b          ;   в регистре CL
        mov     eax, 1              ; создаём маску
        shl     eax, cl             ;   в регистре EAX
        mov     edx, ebx            ; вычисляем номер эл-та
        shr     edx, 5              ;   в регистре edx
        test    [set512+4*edx], eax ; применяем маску
; теперь ZF=1 означает, что элемент в множестве
;   отсутствовал, а ZF=0 - что присутстовал
```

Рассмотрим ещё один пример. Пусть нам потребовалось сосчитать, сколько элементов входят в множество. Для этого придётся просмотреть все элементы массива и в каждом из них сосчитать единичные биты. Проще всего это сделать, загрузив значение из элемента массива в регистр, а потом сдвигая значение вправо на один бит и каждый раз проверяя, единица ли в младшем разряде; это можно делать ровно 32 раза, но проще закончить, когда в регистре останется ноль. Массив мы будем просматривать с конца, индексируя по `ECX`: это позволит нам применить команду `jecxz`. В качестве счётчика результата воспользуемся регистром `EBX`, а для анализа элементов массива применим `EAX`.

```
; сосчитать элементы в множестве set512
        xor     ebx, ebx    ; EBX := 0
        mov     ecx, 15     ; последний индекс
lp:     mov     eax, [set512+4*ecx] ; загрузили элемент
lp2:    test    eax, 1      ; единица в младшем разряде?
        jz      notone      ;     если нет, прыгаем
        inc     ebx         ; если да, увеличиваем счётчик
notone: shr     eax, 1      ; сдвинули EAX
```

```
        test    eax, eax    ; там ещё что-то осталось?
        jnz     lp2         ; если да, продолжаем
                            ;     внутренний цикл
        jecxz   quit        ; если в ECX ноль, заканчиваем
        dec     ecx         ;     иначе уменьшаем его
        jmp     lp          ;     и продолжаем внешний цикл
quit:
; теперь результат подсчёта находится в EBX
```

§ 2.6. Стек, подпрограммы, рекурсия

§ 2.6.1. Понятие стека и его предназначение

Как известно, под *стеком* в программировании подразумевают структуру данных, построенную по принципу «последний вошел — первый вышел» (англ. *last in first out*, LIFO), т. е. такой объект, над которым определены операции «добавить элемент» и «извлечь элемент», причём элементы, которые были добавлены, извлекаются в обратном порядке.

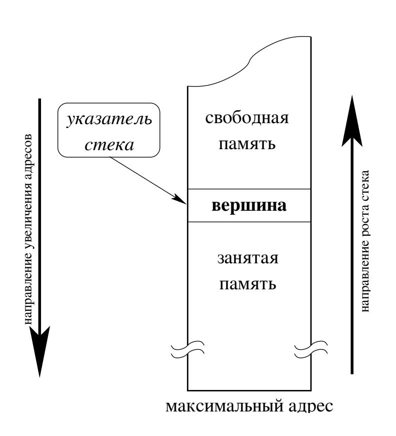


Рис. 2.4. Стек

В применении к низкоуровневому программированию понятие стека существенно у́же: здесь под стеком понимается непрерывная область памяти, для которой в специальном регистре хранится ***адрес вершины стека***, причём память в рассматриваемой области выше вершины (т. е. с адресами, меньшими адреса вершины) считается *свободной*, а память от вершины до конца области (до старших адресов), включая и саму вершину, считается *занятой*; регистр, хранящий адрес вершины, называется ***указателем стека*** (см. рис. 2.4). Операция добавления в стек некоторого значения уменьшает адрес вершины, сдвигая тем самым вершину вверх (то есть в направлении ме́ньших адресов) и в новую вершину записывает добавляемое значение; операция извлечения считывает значение с вершины стека и сдвигает вершину вниз, увеличивая её адрес.

Стек можно использовать, например, для временного хранения значений регистров; если некоторый регистр хранит важное для нас значение, а нам при этом нужно временно задействовать этот регистр для хранения другого значения, то самый простой способ выйти из положения — это сохранить значение регистра в стеке, затем использовать регистр под

другие нужды, и, наконец, восстановить исходное значение регистра путём извлечения этого значения из стека обратно в регистр. Но гораздо более важно другое: **стек используется при вызовах подпрограмм для хранения адресов возврата, для передачи фактических параметров в подпрограммы и для хранения локальных переменных**. Именно использование стека позволяет реализовать механизм рекурсии, при котором подпрограмма может прямо или косвенно вызвать сама себя.

§ 2.6.2. Организация стека в процессоре i386

Большинство существующих процессоров поддерживают работу со стеком на уровне машинных команд, и i386 в этом плане не исключение. Команды работы со стеком позволяют заносить в стек и извлекать из него двухбайтные «слова» и четырёхбайтные «двойные слова»; отдельные байты записывать в стек нельзя, так что адрес вершины стека всегда остаётся чётным.

Как уже говорилось (см. стр. 35), регистр ESP, формально относящийся к группе регистров общего назначения, тем не менее практически никогда не используется ни в какой иной роли, кроме роли ***указателя стека***; название этого регистра как раз и означает «stack pointer». Считается, что адрес, содержащийся в ESP, указывает на вершину стека, то есть на ту область памяти, где хранится последнее занесённое в стек значение. Стек «растёт» в сторону уменьшения адресов, то есть при занесении в стек нового значения ESP уменьшается, при извлечении значения — увеличивается.

Занесение значения в стек производится командой `push`, имеющей один операнд. Этот операнд может быть непосредственным, регистровым или типа «память» и иметь размер `word` или `dword` (если операнд не регистровый, то размер необходимо указать явно). Для извлечения значения из стека используется команда `pop`, операнд которой может быть регистровым или типа «память»; естественно, операнд должен иметь размер «слово» или «двойное слово».

Команды `push` и `pop` совмещают копирование данных (на вершину стека или с неё) со сдвигом самой вершины, то есть изменением значения регистра ESP. Понятно, что можно, вообще говоря, обратиться к значению на вершине стека, не извлекая его из стека — применив (в любой команде, допускающей операнд типа «память») операнд `[esp]`. Например, команда

```
mov eax, [esp]
```

скопирует четырёхбайтное значение с вершины стека в регистр EAX.

Как говорилось выше, стек очень удобно использовать для временного хранения значений из регистров:

```
push eax    ; запоминаем eax
;  ... используем eax под посторонние нужды ...
pop eax     ; восстанавливаем eax
```

Рассмотрим более сложный пример. Пусть регистр ESI содержит адрес некоторой строки символов в памяти, причём известно, что строка заканчивается байтом со значением 0 (но неизвестно, какова длина строки) и нам необходимо «обратить» эту строку, то есть записать составляющие её символы в обратном порядке в том же месте памяти; нулевой байт, играющий роль ограничителя, естественно, остаётся при этом на месте и никуда не копируется. Один из способов сделать это — последовательно записать коды символов в стек, а затем снова пройти строку с начала в конец, извлекая из стека символы и записывая их в ячейки, составляющие строку.

Поскольку записывать в стек однобайтовые значения нельзя, мы будем записывать значения двухбайтовые, причём старший байт просто не будем использовать. Конечно, можно сделать всё более рационально, но нам в данном случае важнее наглядность нашей иллюстрации. Для промежуточного хранения будем использовать регистр BX, причём только его младший байт (BL) будет содержать полезную информацию, но записывать в стек и извлекать из стека мы будем весь BX целиком. Задача будет решена в два цикла. Перед первым циклом мы занесём ноль в регистр ECX, потом на каждом шаге будем извлекать байт по адресу [esi+ecx] и помещать этот байт (в составе слова) в стек, а ECX увеличивать на единицу, и так до тех пор, пока очередной извлечённый байт не окажется нулевым, что по условиям задачи означает конец строки. В итоге все ненулевые элементы строки окажутся в стеке, а в регистре ECX будет длина строки.

Поскольку для второго цикла заранее известно количество его итераций (длина строки) и оно уже содержится в ECX, мы организуем этот цикл с помощью команды loop. Перед входом в цикл мы проверим, не пуста ли строка (то есть не равен ли ECX нулю), и если строка была пуста, сразу же перейдём в конец нашего фрагмента. Поскольку значение в ECX будет уменьшаться, а строку нам нужно пройти в прямом направлении — наряду с ECX мы воспользуемся регистром EDI, который в начале установим равным ESI (то есть указывающим на начало строки), а на каждой итерации будем его сдвигать. Итак, пишем:

```
        xor ebx, ebx        ; обнуляем ebx
        xor ecx, ecx        ; обнуляем ecx
lp:     mov bl, [esi+ecx]   ; очередной байт из строки
```

```
        cmp bl, 0           ; конец строки?
        je lpquit           ; если да - конец цикла
        push bx             ; bl нельзя, приходится bx
        inc ecx             ; следующий индекс
        jmp lp              ; повторить цикл
lpquit: jecxz done          ; если строка пустая - конец
        mov edi, esi        ; опять с начала буфера
lp2:    pop bx              ; извлекаем
        mov [edi], bl       ; записываем
        inc edi             ; следующий адрес
        loop lp2            ; повторять ecx раз
done:
```

§ 2.6.3. Дополнительные команды работы со стеком

При необходимости можно занести в стек значение всех регистров общего назначения одной командой; эта команда называется `pushad` (**push** **a**ll **d**oublewords). Уточним, что эта команда заносит в стек содержимое регистров `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI` и `EDI` (в указанном порядке), причём значение `ESP` заносится в том виде, в котором оно было *до* выполнения команды. Соответствующая команда извлечения из стека называется `popad` (**pop** **a**ll **d**oublewords). Она извлекает из стека восемь четырёхбайтных значений и заносит эти значения в регистры в порядке, обратном приведённому для команды `pushad`, при этом значение, соответствующее регистру `ESP`, игнорируется (то есть из стека извлекается, но в регистр не заносится).

Регистр флагов (EFLAGS) может быть записан в стек командой `pushfd` и считан обратно командой `popfd`, однако при этом, если мы работаем в ограниченном режиме, только некоторые флаги (а именно — флаги, доступные к изменению в ограниченном режиме) могут быть изменены, на остальные команда `popfd` никак не повлияет.

Существуют аналогичные команды для 16-битных регистров, поддерживаемые для совместимости со старыми процессорами; они называются `pushaw`, `popaw`, `pushfw` и `popfw`, и работают полностью аналогично, но вместо 32-битных регистров используют соответствующие 16-битные. Команды `pushaw` и `popaw` практически никогда не используются, что касается команд `pushfw` и `popfw`, то их использование, в принципе, имеет смысл, если учесть, что в «расширенной» части регистра `EFLAGS` нет ни одного флага, значение которого мы могли бы поменять в ограниченном режиме работы.

§ 2.6.4. Подпрограммы: общие принципы

Подпрограммой называется некоторая обособленная часть программного кода, которая может быть *вызвана* из главной программы

(или из другой подпрограммы); под ***вызовом*** в данном случае понимается временная передача управления подпрограмме с тем, чтобы, когда подпрограмма сделает свою работу, она вернула управление в точку, откуда её вызвали. Читатель, вне всякого сомнения, уже встречался с подпрограммами. Это, например, процедуры и функции языка Паскаль, функции в языке Си и т. п.

При вызове подпрограммы необходимо запомнить ***адрес возврата***, то есть адрес машинной команды, следующей за командой вызова подпрограммы, причём сделать это так, чтобы сама вызываемая подпрограмма могла, когда закончит свою работу, воспользоваться этим сохранённым адресом для возврата управления. Кроме того, подпрограммы часто получают ***параметры***, влияющие на их работу, и используют в работе ***локальные переменные***. Подо всё это необходимо предусмотреть выделение оперативной памяти (или регистров). Самый простой способ решения этого вопроса — выделить каждой подпрограмме свою собственную область памяти под хранение всей локальной информации, включая и адрес возврата, и параметры, и локальные переменные. Тогда вызов подпрограммы потребует прежде всего записать в принадлежащую подпрограмме область памяти (в заранее оговорённые места) значения параметров и адрес возврата, а затем передать управление в начало подпрограммы.

Интересно, что когда-то давно именно так с подпрограммами и поступали. Однако с развитием методов и приёмов программирования возникла потребность в ***рекурсии*** — таком построении программы, при котором некоторые подпрограммы могут прямо или косвенно вызывать сами себя, притом потенциально неограниченное[11] число раз. Ясно, что при каждом рекурсивном вызове требуется новый экземпляр области памяти для хранения адреса возврата, параметров и локальных переменных, причём чем позже такой экземпляр будет создан, тем раньше соответствующий вызов закончит работу, то есть рекурсивные вызовы подпрограмм в определённом смысле подчиняются правилу «последний пришел — первый ушел». Совершенно логично из этого вытекает идея использования при вызовах подпрограмм уже знакомого нам стека.

В современных вычислительных системах перед вызовом подпрограммы в стек помещаются значения параметров вызова, затем производится собственно вызов, то есть передача управления, которая совмещена с сохранением в том же стеке адреса возврата. Наконец, когда подпрограмма получает управление, она резервирует в стеке определённое количество памяти для хранения локальных переменных, обычно просто сдвигая адрес вершины вниз на соответствующее количество ячеек. Область стековой памяти, содержащую связанные с одним вызовом зна-

[11] Точнее говоря, ограниченное только объемом памяти.

чения параметров, адрес возврата и локальные переменные, называют ***стековым фреймом***.

§ 2.6.5. Вызов подпрограмм и возврат из них

Вызов подпрограммы, как уже стало ясно из вышесказанного,— это передача управления по адресу начала подпрограммы с одновременным запоминанием в стеке адреса возврата (то есть адреса машинной команды, непосредственно следующей за командой вызова). Процессор i386 предусматривает для этой цели команду `call`; аналогично команде `jmp`, аргумент команды `call` может быть непосредственным (адрес перехода задан непосредственно в команде, например, меткой), регистровым (адрес передачи управления находится в регистре) и типа «память» (переход нужно осуществить по адресу, прочитанному из заданного места памяти). Команда `call` не имеет «короткой» формы; поскольку «дальняя» форма нам, как обычно, не требуется в силу отсутствия сегментов, остаётся только одна форма — близкая (`near`), которую мы всегда и используем.

Возврат из подпрограммы производится командой `ret` (от слова return). В своей простейшей форме эта команда не имеет аргументов. Выполняя эту команду, процессор извлекает 4 байта с вершины стека и записывает их в регистр `EIP`, в результате чего управление передаётся по адресу, который находился в памяти на вершине стека.

Рассмотрим простой пример. Допустим, в нашей программе часто приходится заполнять каким-то однобайтовым значением области памяти разной длины. Такое действие вполне можно оформить в виде подпрограммы. Для простоты картины примем соглашение, что адрес нужной области памяти передаётся через регистр `EDI`, количество однобайтовых ячеек, которые нужно заполнить — через регистр `ECX`, ну а само значение, которое надо записать во все эти ячейки — через регистр `AL`. Код соответствующей подпрограммы может выглядеть, например, так:

```
; fill memory (edi=address, ecx=length, al=value)
fill_memory:
        jecxz   fm_q
fm_lp:  mov     [edi], al
        inc edi
        loop fm_lp
fm_q:   ret
```

Обратиться к такой подпрограмме можно, например, так:

```
        mov edi, my_array
        mov ecx, 256
```

```
mov al, '@'
call fill_memory
```

В результате такого вызова 256 байт памяти, начиная с адреса, заданного меткой my_array, окажутся заполнены кодом символа '@' (число 64).

§ 2.6.6. Организация стековых фреймов

Подпрограмма, приведённая в качестве примера в предыдущем параграфе, фактически не использовала механизм стековых фреймов, сохраняя в стеке только адрес возврата. Этого оказалось достаточно, поскольку подпрограмме не требовались локальные переменные, а параметры мы передали через регистры. Как показывает практика, подпрограммы редко бывают такими простыми. В более сложных случаях нам наверняка потребуются локальные переменные, поскольку регистров на всё не хватит. Кроме того, передача параметров через регистры тоже может оказаться неудобна: во-первых, регистров может и не хватить, а во-вторых, подпрограмме могут быть долго нужны значения, переданные через регистры, и это фактически лишит её возможности использовать под свои внутренние нужды те из регистров, которые были задействованы при передаче параметров. Наконец, передача параметров через регистры (а равно и через какую-либо фиксированную область памяти) лишает нас возможности использовать рекурсию, что тоже, разумеется, плохо.

Поэтому обычно (в особенности при трансляции программы с какого-либо языка высокого уровня, с того же Паскаля или Си) параметры в функции передаются через стек, и в стеке же размещаются локальные переменные. Как было сказано выше, параметры в стеке размещает вызывающая программа, затем при вызове подпрограммы в стек заносится адрес возврата, и, наконец, уже сама вызванная подпрограмма резервирует место в стеке под локальные переменные. Всё это вместе и образует стековый фрейм. К содержимому стекового фрейма можно обращаться, используя адреса, «привязанные» к адресу, по которому содержится адрес возврата, то есть, иначе говоря, ту ячейку памяти, начиная с которой в стек был занесён адрес возврата, используют в качестве своего рода реперной точки. Так, если в стек занести три четырёхбайтных параметра, а потом вызвать процедуру, то адрес возврата будет лежать в памяти по адресу [esp], ну а параметры, очевидно, окажутся доступны по адресам [esp+4], [esp+8] и [esp+12]. Если же разместить в стеке локальные четырёхбайтные переменные, то они окажутся доступны по адресам [esp-4], [esp-8] и т. д.

Заметим, что использовать для доступа к параметрам регистр ESP оказывается не слишком удобно, ведь в самой процедуре нам тоже может потребоваться стек (как для временного хранения данных, так и

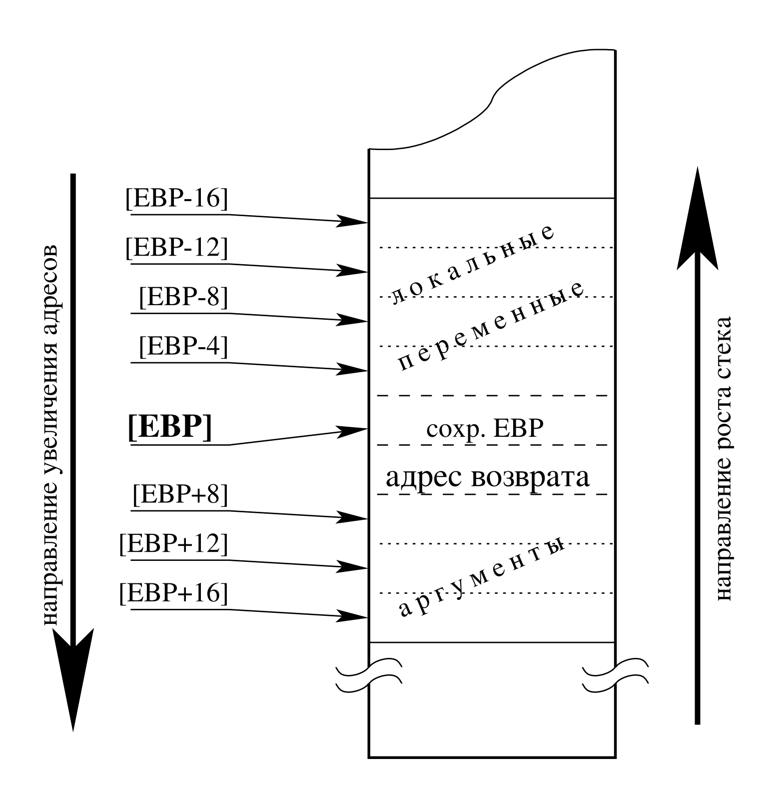


Рис. 2.5. Структура стекового фрейма

для вызова других подпрограмм). Поэтому первым же своим действием подпрограмма обычно сохраняет значение регистра ESP в каком-то другом регистре (чаще всего EBP) и именно его использует для доступа к параметрам и локальным переменным, ну а регистр ESP продолжает играть свою роль указателя стека, изменяясь по мере необходимости; перед возвратом из подпрограммы его обычно восстанавливают в исходном значении (попросту пересылая в него значение из EBP), чтобы он снова указывал на адрес возврата.

Наконец, возникает ещё один вопрос: а что если другие подпрограммы тоже используют регистр EBP для тех же целей? Ведь в этом случае первый же вызов другой подпрограммы испортит нам всю работу. Можно, конечно, сохранять EBP в стеке перед вызовом каждой подпрограммы, но поскольку в программе обычно гораздо больше *вызовов подпрограмм*, чем собственно самих подпрограмм, экономнее оказывается следовать простому правилу: *каждая подпрограмма должна сама сохранить старое значение EBP и восстановить его перед возвратом управления.* Естественно, для сохранения значения EBP тоже используется стек, причём

сохранение выполняется простой командой push ebp сразу после получения управления. Таким образом, старое значение EBP помещается в стек непосредственно после адреса возврата из подпрограммы, и в качестве «точки привязки» используется в дальнейшем именно этот адрес вершины стека. Для этого следующей командой выполняется mov ebp,esp. В результате регистр EBP указывает на то место в стеке, где находится его же, EBP, сохранённое значение; если теперь обратиться к памяти по адресу [ebp+4], мы обнаружим там адрес возврата из подпрограммы, ну а параметры, занесённые в стек перед вызовом подпрограммы, оказываются доступны по адресам [esp+8], [esp+12], [esp+16] и т. д. Память под локальные переменные выделяется путём простого вычитания нужной длины из текущего значения ESP; так, если под локальные переменные нам нужно 16 байт, то сразу после сохранения EBP и копирования в него содержимого ESP нужно выполнить команду sub esp, 16; если (для простоты картины) все наши локальные переменные тоже занимают по 4 байта, они окажутся доступны по адресам [ebp-4], [ebp-8] и т. д. Структура стекового фрейма с тремя четырёхбайтными параметрами и четырьмя четырёхбайтными локальными переменными показана на рис. 2.5.

Повторим, что в начале своей работы, согласно нашим договорённостям, каждая подпрограмма должна выполнить

```
push ebp
mov ebp, esp
sub esp, 16     ; вместо 16 подставьте объем
                ; памяти под локальные переменные
```

а завершение подпрограммы теперь должно выглядеть так:

```
mov esp, ebp
pop ebp
ret
```

Интересно, что процессор i386 поддерживает даже специальные команды для обслуживания стековых фреймов. Так, в начале подпрограммы вместо трёх команд, приведённых выше, можно было бы дать одну команду

```
enter 16, 0
```

а вместо двух команд перед ret можно было бы написать

```
leave
```

Проблема, как ни странно, в том, что команды enter и leave работают *медленнее*, чем соответствующий набор простых команд, так что их практически никогда не используют; если дизассемблировать машинный код, сгенерированный компилятором языка Си или Паскаль, мы, скорее всего, обнаружим в начале любой

процедуры или функции именно такие команды, как показано выше, и ничего похожего на enter. Единственным оправданием существования команд enter и leave может служить их короткая запись (например, машинная команда leave занимает в памяти всего 1 байт), но в наше время об экономии памяти на машинном коде обычно никто не задумывается; быстродействие практически всегда оказывается важнее.

Сделаем ещё одно важное замечание. При работе под управлением ОС Unix мы можем не беспокоиться ни о наличии стека, ни о задании его размера. Операционная система создаёт стек автоматически при запуске любой задачи и, более того, уже во время её исполнения при необходимости увеличивает размер доступной для стека памяти: по мере того как вершина стека продвигается по виртуальному адресному пространству «вверх» (то есть в сторону уменьшения адресов), операционная система ставит в соответствие виртуальным адресам всё новые и новые страницы физической памяти. Именно поэтому на рис. 2.4 и 2.5 мы изобразили верхний край стека как нечто нечёткое.

§ 2.6.7. Основные конвенции вызовов подпрограмм

Несмотря на подробное описание механизма стековых фреймов, данное в предыдущем параграфе, в некоторых вопросах остаётся возможность для манёвра. Так, например, в каком порядке следует заносить в стек значения фактических параметров? Если мы пишем программу на языке ассемблера, этот вопрос, собственно говоря, не встаёт; однако он оказывается неожиданно принципиальным при создании компиляторов языков программирования высокого уровня.

Создатели компиляторов языка Паскаль обычно идут «очевидным» путём: вызов процедуры или функции транслируется с Паскаля в виде серии команд занесения в стек значений, причём значения заносятся в естественном (для человека) порядке — слева направо; затем в код вставляется команда call. Когда такая процедура получает управление, значения фактических параметров располагаются в стеке снизу вверх, то есть *последний* параметр оказывается размещён ближе других к реперной точке фрейма (доступен по адресу [ebp+8]). Из этого, в свою очередь, следует, что **для доступа к первому (а равно и к любому другому) фактическому параметру процедуре или функции языка Паскаль необходимо знать общее количество этих параметров**, поскольку расположение n-го параметра в стековом фрейме получается зависящим от общего количества. Так, если у процедуры три четырёхбайтных параметра, то первый из них окажется в стеке по адресу [ebp+16], если же их пять, то первый придётся искать по адресу [ebp+24]. Именно поэтому язык Паскаль не допускает создание процедур или функций с переменным числом аргументов, так называемых ***вариадических*** подпрограмм

(что вполне нормально для учебного языка, но не совсем приемлемо для языка профессионального). Входящие в язык Паскаль псевдопроцедуры с переменным числом аргументов, такие как `WriteLn`, на самом деле являются частью самого языка Паскаль; компилятор трансформирует их вызовы в нечто весьма далёкое от вызова подпрограммы на уровне машинного кода. Так или иначе, программист не может на Паскале описать свою процедуру подобного рода.

Создатели языка Си пошли иным путём. При трансляции вызова функции языка Си параметры помещаются в стек *в обратном порядке* и оказываются размещёнными во фрейме в порядке сверху вниз, так что первый параметр всегда оказывается доступен по адресу `[ebp+8]`, второй — по адресу `[ebp+12]` и т. д., вне всякой зависимости от общего количества параметров (конечно, параметры по крайней мере должны присутствовать, то есть если функция, например, вызвана вообще без параметров, никакого первого параметра в стеке не будет). Это, с одной стороны, позволяет создание вариадических функций; в частности, в сам по себе язык Си не входит вообще ни одной функции, что же касается таких функций, как `printf`, `scanf` и др., то они реализуются в *библиотеке*, а не в самом языке, и, более того, сами эти функции тоже написаны на Си (как сказано выше, на Паскале так сделать не получается).

С другой стороны, отсутствие в Паскале вариадических подпрограмм позволяет возложить заботы об очистке стека на *вызываемого*. Действительно, подпрограмма языка Паскаль всегда знает, сколько места занимают фактические параметры в её стековом фрейме (поскольку для каждой подпрограммы это количество задано раз и навсегда и не может измениться) и, соответственно, может принять на себя заботу об очистке стека. Как уже говорилось, вызовов подпрограмм в любой программе больше, чам самих подпрограмм, так что за счёт перекладывания заботы об очистке стека с вызывающего на вызываемого достигается определённая экономия памяти (количества машинных команд). При использовании соглашений языка Си такая экономия невозможна, поскольку подпрограмма не знает и не может знать (в общем случае[12]), сколько параметров ей передали, так что забота об очистке стека от параметров остаётся на вызывающем; обычно это делается простым увеличением значения `ESP` на число, равное совокупной длине фактических параметров. Например, если подпрограмма `proc1` принимает на вход три четырёхбайтных параметра (назовём их `a1`, `a2` и `a3`), её вызов будет выглядеть примерно так:

```
push dword a3  ; заносим в стек параметры
```

[12]В разных ситуациях используются различные способы фиксации количества параметров; так, функция `printf` узнаёт, сколько параметров нужно извлечь из стека, путём анализа форматной строки, а функция `execlp` извлекает аргументы, пока не наткнётся на нулевой указатель, но и то и другое — лишь частные случаи.

```
        push dword a2
        push dword a1
        call proc1      ; вызываем подпрограмму
        add esp, 12     ; убираем параметры из стека
```

В случае же использования соглашений языка Паскаль последняя команда (`add`) оказывается не нужна, обо всём позаботится вызываемый. Процессор i386 даже имеет для этого специальную форму команды `ret` с одним операндом (выше в примерах мы использовали `ret` без операндов). Этот операнд, который может быть только непосредственным и всегда имеет длину два байта («слово»), задаёт количество памяти (в байтах), занятой параметрами функции. Например, процедуру, принимающую три четырёхбайтных параметра, комиплятор Паскаля закончит командой

```
        ret 12
```

Эта команда, как и обычная команда `ret`, извлечёт из стека адрес возврата и передаст по нему управление, но кроме этого (одновременно с этим) увеличит значение `ESP` на заданное число (в данном случае 12), избавляя, таким образом, вызвавшего от обязанности по очистке стека.

§ 2.6.8. Локальные метки

Прежде чем мы приведём пример подпрограммы, выполняющей рекурсивный вызов, необходимо рассмотреть ещё одно важное средство, предоставляемое ассемблером NASM — ***локальные метки***.

Суть и основное достоинство подпрограмм состоит в их *обособленности*. Иначе говоря, в процессе написания одной подпрограммы мы обычно не помним, как изнутри устроены другие подпрограммы и воспринимаем каждую из подпрограмм, кроме одной (той, что пишется прямо сейчас) в виде этакой одной большой команды. Это позволяет не держать в голове лишних деталей и сосредоточиться на реализации конкретного фрагмента программы, а по окончании такой реализации выкинуть её детали из головы и перейти к другому фрагменту.

Проблема состоит в том, что в теле любой сколь бы то ни было сложной подпрограммы нам обязательно понадобятся метки, и нужно сделать так, чтобы при выборе имён для таких меток нам не нужно было вспоминать, есть ли уже где-нибудь (в другой подпрограмме) метка с таким же именем.

Ассемблер NASM для этого предусматривает специальные *локальные* метки. Синтаксически эти метки отличаются от обычных тем, что начинаются с точки. Ассемблер локализует такие метки во фрагменте программы, ограниченном с обеих сторон обычными (нелокальными) метками. Иначе говоря, локальную метку ассемблер рассматривает не саму

по себе, а как нечто подчинённое последней (ближайшей сверху) нелокальной метке. Например, в следующем фрагменте:

```
first_proc:
        ; ... ...
.cycle:
        ; ... ...
second_proc:
        ; ... ...
.cycle:
        ; ... ...
third_proc:
```

первая метка `.cycle` подчинена метке `first_proc`, а вторая — метке `second_proc`, так что между собой они не конфликтуют. Если метка `.cycle` встретится в параметрах той или иной команды между метками `first_proc` и `second_proc`, ассемблер будет знать, что имеется в виду именно первая из меток `.cycle`, если она встретится после `second_proc`, но перед `third_proc` — то задействуется вторая, тогда как появление метки `.cycle` до `first_proc` или после `third_proc` будет рассматриваться как ошибка. Таким образом, если каждую подпрограмму начинать с обычной метки, а внутри подпрограммы использовать только локальные метки, то в разных подпрограммах мы можем использовать локальные метки с одинаковыми именами, и ассемблер в них не запутается.

На самом деле, ассемблер достигает такого эффекта за счёт «не очень честного» приёма — видя метку, имя которой начинается с точки, он просто добавляет к ней спереди имя последней встречавшейся ему метки без точки. Таким образом, в примере выше речь идёт не о двух одинаковых метках `.cycle`, а о двух *разных* метках `first_proc.cycle` и `second_proc.cycle`. Полезно помнить об этом и не применять в программе в явном виде метки, содержащие точку, несмотря на то, что ассемблер это допускает.

§ 2.6.9. Пример

Приведём пример подпрограммы, использующей рекурсию. Одна из простейших классических задач, решаемых рекурсивно — это сопоставление строки с образцом, её мы и используем в примере.

Для начала уточним задачу. Даны две строки символов, длина которых заранее неизвестна, но известно, что каждая из них ограничена нулевым байтом. Первую строку мы рассматриваем как *сопоставляемую*, вторую воспринимаем как *образец*. В образце символ `'?'` может сопоставляться с произвольным символом, символ `'*'` — с произвольной *подцепочкой символов* (возможно даже пустой), остальные символы обозначают сами себя и только сами с собой сопоставляются. Так, образцу `'abc'` соответствует только строка `'abc'`; образцу `'a?c'` соответствует

любая строка из трёх символов, начинающаяся на 'a' и заканчивающаяся на 'c' (символ в середине может быть любым). Наконец, образцу 'a*' соответствует любая строка, начинающаяся на 'a', ну а образцу '*a*' соответствует любая строка, содержащая букву 'a' в любом месте. Необходимо определить, соответствует ли (целиком) заданная строка заданному образцу, и вернуть результат 0, если не соответствует, и результат 1, если соответствует.

Алгоритм такого сопоставления, если при этом можно использовать рекурсию, окажется достаточно простым. На каждом шаге мы рассматриваем *оставшуюся часть* строки и образца; сначала эти оставшиеся части совпадают со строкой и образцом, затем, по мере продвижения алгоритма, от них отбрасываются символы, стоящие в начале, и мы предполагаем, что для уже отброшенных символов сопоставление прошло успешно. Первое, что нужно сделать в начале каждого шага — это проверить, не кончился ли у нас образец. Если он кончился, то результат зависит от того, кончилась ли при этом и строка тоже. Если кончилась, то мы возвращаем единицу (истину), если не кончилась — возвращаем ноль (ложь); действительно, с пустым образцом можно сопоставить только пустую строку.

Если образец ещё не кончился, проверяем, не находится ли в начале него (то есть в первом символе остатка образца) символ '*'. Если нет, то всё просто: мы производим сопоставление первых символов строки и образца; если первый символ образца не является символом '?' и не равен первому символу строки, то алгоритм на этом завершается и мы возвращаем ложь, в противном случае считаем, что очередные символы образца и строки успешно сопоставлены, отбрасываем их (то есть укорачиваем остатки обеих строк спереди) и возвращаемся к началу алгоритма.

Самое интересное происходит, если на очередном шаге первый символ образца оказался символом '*'. В этом случае нам нужно последовательно перебрать возможности сопоставления этой «звёздочки» с пустой подцепочкой строки, с одним символом строки, с двумя символами и т. д., пока не кончится сама строка. Делаем мы это следующим образом. Заводим целочисленную переменную I, которая будет у нас обозначать текущий рассматриваемый вариант. Присваиваем этой переменной ноль (начинаем рассмотрение с пустой цепочки). Теперь для каждой рассматриваемой альтернативы отбрасываем от образца один символ (звёздочку), а от строки — столько символов, какое сейчас число в переменной I. Полученные остатки *пытаемся сопоставить, используя для этого вызов той самой подпрограммы, которую мы сейчас пишем*, то есть производим рекурсивный вызов «самих себя». Если результат вызова — истина, то мы на этом завершаемся, тоже вернув истину. Если же результат — ложь, то мы проверяем, можно ли ещё увеличивать переменную I (не

вылетим ли мы при этом за пределы сопоставляемой строки). Если увеличиваться уже некуда, завершаем работу, вернув ложь. В противном случае возвращаемся к началу цикла и рассматриваем следующее возможное значение I.

Для читателей, знакомых с языком программирования Си, отметим, что на этом языке вышеописанный алгоритм может быть реализован следующей функцией:

```
int match(const char *str, const char *pat)
{
    int i;
    for(;; str++, pat++) {
        switch(*pat) {
            case 0:
                return *str == 0;
            case '*':
                for(i=0; ; i++) {
                    if(match(str+i, pat+1)) return 1;
                    if(!str[i]) return 0;
                }
            case '?':
                if(!*str) return 0;
                break;
            default:
                if(*str != *pat) return 0;
        }
    }
}
```

На Паскале такая же функция будет выглядеть несколько более громоздко. Причиной тому, во-первых, отсутствие в Паскале арифметики указателей, и во-вторых, приницпиально другой подход к работе со строками, который в данной конкретной задаче менее удобен (хотя во многих других задачах оказывается удобнее, чем подход, традиционный для Си). Вот пример реализации:

```
function match(str, pat: string): boolean;
  function do_match(str_ind, pat_ind: integer): boolean;
    var i: integer;
  begin
    while true do begin
      if pat_ind > length(pat) then begin
        do_match := str_ind > length(str); exit
      end;
      if pat[pat_ind] = '*' then begin
        for i:=0 to length(str)-str_ind+1 do
          if do_match(str_ind+i, pat_ind+1) then begin
            do_match := true; exit
          end;
```

```
        do_match := false; exit
      end else
      if (str_ind > length(str)) or
         ((str[str_ind] <> pat[pat_ind]) and
          (pat[pat_ind] <> '?'))
      then begin
        do_match := false; exit
      end;
      str_ind := str_ind + 1;
      pat_ind := pat_ind + 1;
    end
  end;
begin
  match := do_match(1,1)
end;
```

Передав в функцию match строки, подлежащие сопоставлению, мы дальше меняем только индексы; чтобы организовать по ним рекурсию, мы описали локальную функцию do_match, которая и выполняет всю работу.

Реализацию на языке ассемблера мы выполним в виде подпрограммы, которую назовём match. Подпрограмма будет предполагать, что ей передано два параметра — адрес строки ([ebp+8]) и адрес образца ([ebp+12]); сама подпрограмма будет использовать одну четырёхбайтную переменную для хранения I; под неё будет выделяться место в стековом фрейме и она будет, соответственно, располагаться по адресу [ebp-4]. Для увеличения скорости работы наша подпрограмма в самом начале скопирует адреса из параметров в регистры ESI (адрес строки) и EDI (адрес образца). Кроме того, для выполнения арифметических действий наша подпрограмма будет использовать регистр EAX. Через него же подпрограмма будет возвращать результат своей работы: число 0 как обозначение логической лжи (соответствие не найдено) или число 1 как обозначение логической истины (соответствие найдено).

«Отбрасывание» символов из начала строк мы будем производить простым увеличением рассматриваемого адреса строки: действительно, если по адресу string находится строка, мы можем считать, что по адресу string+1 находится та же строка, кроме первой буквы.

Необходимо обратить внимание, что подпрограмма будет рекурсивно вызывать сама себя, и, будучи вызванной рекурсивно, должна будет выполнять работу над значениями, отличающимися от тех, что были заданы в предыдущем вызове. Между тем, регистры в качестве хранилища локальных данных понадобятся как исходному вызову подпрограммы, так и «вложенному» (рекурсивному), но в процессоре только один набор регистров. Мы выйдем из положения вполне традиционным способом: наша функция будет перед началом работы сохранять в стеке не только EBP, но и все остальные регистры, которые она использует, а перед воз-

вратом управления регистры будут восстанавливаться; в данном случае мы используем ESI, EDI и EAX, но EAX в любом случае «испортится», поскольку через него мы возвращаем итоговое значение, так что сохранять нужно только ESI и EDI. Так обычно действуют не только в случае рекурсии, но и вообще в любых подпрограммах: этот подход позволяет в любой из наших подпрограмм не думать о том, что подпрограммы, к которым мы обращаемся, могут испортить регистры, где хранятся нужные нам значения.

```
match:                          ; НАЧАЛО ПОДПРОГРАММЫ
        push ebp                ; организуем стековый фрейм
        mov ebp, esp
        sub esp, 4              ; локальная переменная I
                                ; будет по адресу [ebp-4]
        push esi                ; сохраняем регистры ESI и EDI
        push edi                ;     (EAX всё равно изменится)
        mov esi, [ebp+8]        ; загружаем параметры: адреса
        mov edi, [ebp+12]       ;     строки и образца
.again:                         ; сюда мы вернёмся, когда
                                ;     сопоставим очередной
                                ;     символ и сдвинемся
        cmp byte [edi], 0       ; образец кончился?
        jne .not_end            ;     если нет, прыгаем
        cmp byte [esi], 0       ; образец кончился, а строка?
        jne near .false         ; если нет, то вернуть ЛОЖЬ
        jmp .true               ; кончились одновременно: ИСТИНА
.not_end:                       ; если образец не кончился...
        cmp byte [edi], '*'     ; не звёздочка ли в его начале?
        jne .not_star           ; если нет, прыгаем отсюда
                                ; звёздочка! организуем цикл
        mov dword [ebp-4], 0    ; I := 0
.star_loop:
                                ; готовимся к рекурс. вызову
        mov eax, edi            ; сначала второй аргумент:
        inc eax                 ;      образец со след. символа
        push eax
        mov eax, esi            ; теперь первый аргумент:
        add eax, [ebp-4]        ;      строка с I-го символа
        push eax                ;  (напомним, [ebp-4] - это I)
        call match              ; вызываем сами себя, но
                                ;      с новыми параметрами
        add esp, 8              ; после вызова очищаем стек
        test eax, eax           ; что нам вернули?
        jnz .true               ; Вернули не ноль, т.е. ИСТИНУ
                                ; Значит, остаток строки
                                ; сопоставился с остатком
                                ; образца => вернём истину
```

```
        add eax, [ebp-4]      ; вернули 0, т.е. ЛОЖЬ
                              ; надо попробовать больше
                              ; символов "списать" на
                              ; эту звёздочку
        cmp byte [esi+eax], 0 ; Но, быть может, строка
                              ;      уже кончилась?
        je .false             ; Тогда пробовать больше нечего
        inc dword [ebp-4]     ; Иначе пробуем: I := I + 1
        jmp .star_loop        ;      и в начало цикла по I

.not_star:                    ; сюда мы попадаем, если обр.
        mov al, [edi]         ;      не пуст и не нач. с '*'
        cmp al, '?'           ; может быть, там знак '?'
        je .quest             ; если да, прыгаем отсюда
        cmp al, [esi]         ; если нет, символы в начале
                              ;      строки и образца должны
                              ;      совпадать; если строка
                              ;      кончилась, эта проверка
                              ;      тоже не пройдёт
        jne .false            ; Не совпали (или кон. строки)
                              ;      => возвращаем ЛОЖЬ
        jmp .goon             ; Совпали - продолжаем
                              ;      просмотр
.quest:                       ; Образец начинается с '?'
        cmp byte [esi], 0     ; Надо только, чтобы строка не
        jz .false             ;      кончилась (иначе ЛОЖЬ)
.goon:  inc esi               ; Символы сопоставились =>
        inc edi               ;      сдвигаемся по строке и
        jmp .again            ;      образцу и продолжаем
.true:                        ; Сюда мы прыгали, чтобы
        mov eax, 1            ;      вернуть ИСТИНУ
        jmp .quit
.false:                       ; А сюда прыгали, чтобы
        xor eax, eax          ;      вернуть ЛОЖЬ
.quit:                        ; Всё, конец работы
        pop edi               ;      Приводим всё в
        pop esi               ;      порядок перед
        mov esp, ebp          ;      возвратом управления
        pop ebp               ; Результат у нас в EAX
        ret                   ; Возвращаем управление
                              ; КОНЕЦ ПРОЦЕДУРЫ
```

Если, например, ваша строка располагается в памяти, помеченной меткой `string`, а образец — в памяти, помеченной меткой `pattern`, то вызов подпрограммы `match` будет выглядеть вот так:

```
        push dword pattern
```

```
push dword string
call match
add esp, 8
```

После этого результат сопоставления (0 или 1) окажется в регистре EAX.

Обратите внимание, что в начале подпрограммы при попытке перейти на метку `.false` мы были вынуждены явно указать, что переход является «ближним» (`near`). Дело в том, что метка `.false` оказалась чуть дальше от команды перехода, чем это допустимо для «короткого» перехода. См. обсуждение на стр. 60.

§ 2.7. Строковые операции

Для упрощения выполнения действий над массивами (непрерывными областями памяти) процессор i386 вводит несколько команд, объединяемых в категорию ***строковых операций***. Именно эти команды используют регистры ESI и EDI в их особой роли, обсуждавшейся на стр. 35. Общая идея строковых команд состоит в том, что чтение из памяти выполняется по адресу из регистра ESI, запись в память — по адресу из регистра EDI, а затем эти регистры увеличиваются (или уменьшаются) в зависимости от команды на 1, 2 или 4. Некоторые команды производят чтение в регистр или запись в память из регистра; в этом случае используется регистр «аккумулятор» соответствующего размера, то есть регистр AL, AX или EAX. Строковые команды не имеют операндов, всегда используя одни и те же регистры.

«Направление» изменения адресов (движения вдоль строк) определяется флагом DF (напомним, его имя означает «direction flag», т. е. «флаг направления»). Если этот флаг сброшен, адреса увеличиваются (то есть строковая операция выполняется слева направо), если флаг установлен — адреса уменьшаются (соответственно, работаем справа налево). Установить DF можно командой `std` (**s**e**t** **d**irection), а сбросить — командой `cld` (**cl**ear **d**irection).

Самые простые из строковых команд — команды `stosb`, `stosw` и `stosd`, которые записывают в память по адресу `[edi]`, соответственно, байт, слово или двойное слово из регистра AL, AX или EAX, после чего увеличивают или уменьшают (в зависимости от значения DF) регистр EDI на 1, 2 или 4. Например, если у нас есть массив

```
buf     resb 1024
```

и нам нужно заполнить его нулями, мы можем применить следующий код:

```
xor al, al     ; обнуляем al
mov edi, buf   ; адрес начала массива
```

```
        mov ecx, 1024 ; длина массива
        cld           ; работаем в прямом направлении
lp:     stosb         ; al -> [edi], увел. edi
        loop lp
```

Эти и другие строковые команды удобно использовать с ***префиксом*** rep. Команда, снабженная таким префиксом, будет выполнена столько раз, какое число было в регистре ECX (кроме команды stosw: если её снабдить префиксом, то будет использоваться регистр CX; это обусловлено историческими причинами). С помощью префикса rep мы можем переписать вышеприведённый пример без использования метки:

```
        xor al, al
        mov edi, buf
        mov ecx, 1024
        cld
        rep stosb
```

Команды lodsb, lodsw и lodsd, наоборот, считывают байт, слово или двойное слово из памяти по адресу, находящемуся в регистре ESI, и помещают прочитанное в регистр AL, AX или EAX, после чего увеличивают или уменьшают значение регистра ESI на 1, 2 или 4. Использование этих команд с префиксом rep обычно бессмысленно, поскольку мы не сможем между последовательными исполнениями строковой команды вставить какие-то ещё действия, обрабатывающие значение, прочитанное и помещённое в регистр. Однако использование команд серии lods без префикса может оказаться весьма полезным. Пусть, например, у нас есть массив четырёхбайтных чисел

```
array   resd 256
```

и нам необходимо сосчитать сумму его элементов. Это можно сделать следующим образом:

```
        xor ebx, ebx   ; зануляем сумму
        mov esi, array
        mov ecx, 256
        cld
lp:     lodsd
        add ebx, eax
        loop lp
```

Часто оказывается удобным сочетание команд серии lods с соответствующими командами stos. Пусть, например, нам нужно увеличить на единицу все элементы того же самого массива. Это можно сделать так:

```
        mov esi, array
        mov edi, esi
        mov ecx, 256
        cld
lp:     lodsd
        inc eax
        stosd
        loop lp
```

Если же необходимо просто скопировать данные из одной области памяти в другую, очень удобны оказываются команды `movsb`, `movsw` и `movsd`. Эти команды копируют байт, слово или двойное слово из памяти по адресу `[esi]` в память по адресу `[edi]`, после чего увеличивают (или уменьшают) сразу оба регистра `ESI` и `EDI` (соответственно, на 1, 2 или 4). Например, если у нас есть два строковых массива

```
buf1    resb 1024
buf2    resb 1024
```

и нужно скопировать содержимое одного из них в другой, можно сделать это так:

```
        mov ecx, 1024
        mov esi, buf1
        mov edi, buf2
        cld
        rep movsb
```

Благодаря возможности изменять направление работы (с помощью `DF`), мы можем производить копирование *частично перекрывающихся* областей памяти. Пусть, например, в массиве `buf1` содержится строка `"This is a string"` и нам нужно перед словом `"string"` вставить слово `"long"`. Для этого сначала нужно скопировать область памяти, начиная с адреса `[buf1+10]`, на пять байт вперёд, чтобы освободить место для слова `"long"` и пробела. Ясно, что производить такое копирование мы можем только из конца в начало, иначе часть букв будет затёрта до того, как мы их скопируем. Таким образом, если слово `"long "` (вместе с пробелом) содержится в буфере `buf2`, то вставить его во фразу, находящуюся в `buf1`, мы можем так:

```
        std
        mov edi, buf1+17+5
        mov esi, buf1+17
        mov ecx, 8
        rep movsb
```

```
mov esi, buf2+4
mov ecx, 5
rep movsb
```

Кроме перечисленных, процессор i386 реализует команды `cmpsb`, `cmpsw` и `cmpsd` (**comp**are **s**tring), а также `scasb`, `scasw` и `scasd` (**sca**n **s**tring). Команды серии `scas` сравнивают аккумулятор (соответственно, `AL`, `AX` или `EAX`) с байтом, словом или двойным словом по адресу `[edi]`, устанавливая соответствующие флаги подобно команде `cmp`, и увеличивают/уменьшают `EDI` Команды серии `cmps` сравнивают байты, слова или двойные слова, находящиеся в памяти по адресам `[esi]` и `[edi]`, устанавливают флаги и увеличивают/уменьшают оба регистра.

Кроме префикса `rep`, можно воспользоваться также префиксами `repz` и `repnz` (также называемыми `repe` и `repne`), которые, кроме уменьшения и проверки регистра `ECX` (или `CX`, если команда двухбайтная) также проверяют значение флага `ZF` и продолжают работу, только если этот флаг установлен (`repz`/`repe`) или сброшен (`repnz`/`repne`). Обычно эти префиксы используют как раз в сочетании с командами серий `scas` и `cmps`.

§ 2.8. Ещё несколько интересных команд

В завершение изучения системы команд процессора i386 рассмотрим ещё несколько команд.

Команды `cbw`, `cwd`, `cwde` и `cdq` предназначены для *увеличения разрядности знакового числа*; попросту говоря, они заполняют дополнительные разряды значением знакового бита исходного числа. Все эти четыре команды не имеют операндов и всегда работают с одними и теми же регистрами. Команда `cbw` расширяет число в регистре `AL` до всего регистра `AX`, т. е. заполняет разряды регистра `AH`. Команда `cwd` расширяет число в регистре `AX` до регистровой пары `DX:AX`, то есть заполняет разряды регистра `DX`. Команда `cwde` расширяет тот же регистр `AX` до регистра `EAX`, заполняя старшие 16 разрядов этого регистра. Наконец, команда `cdq` расширяет `EAX` до регистровой пары `EDX:EAX`, заполняя разряды регистра `EDX`. Особенно актуальными эти команды оказываются в сочетании с командой целочисленного деления (`div`, см. § 2.3.4).

Команды `movsx` (**mov**e **s**igned e**x**tension) и `movzx` (**mov**e **z**ero e**x**tension) позволяют совместить копирование с увеличением разрядности. Обе команды имеют по два операнда, причём первый операнд обязан быть регистровым, а второй может быть регистром или памятью, и в любом случае длина первого операнда должна быть вдвое больше длины второго (то есть можно копировать из байта в слово или из слова в двойное сло-

во). Недостающие разряды команда movzx заполняет нулями, а команда movsx — значением старшего бита исходного операнда.

Наконец, рассмотрение системы команд не может считаться законченным без команды nop. Она выполняет очень важное действие: *не делает ничего*. Само её название образовано от слов «No OPeration».

§ 2.9. Заключительные замечания

Конечно, мы не рассмотрели и десятой доли возможностей процессора i386, если же говорить о расширениях его возможностей, появившихся в более поздних процессорах (например, MMX-регистры), то доля изученного нами окажется ещё скромнее. Однако *писать программы на языке ассемблера* мы теперь можем, и это позволит нам получить опыт программирования в терминах машинных команд, что, как было сказано в предисловии, является необходимым условием качественного программирования *вообще на любом языке программирования*: нельзя создавать хорошие программы, не понимая, что на самом деле происходит.

Читатели, у которых возникнет желание изучить аппаратную платформу i386 более глубоко, могут обратиться к технической документации и справочникам, которые в более чем достаточном количестве представлены в сети Интернет. Хочется, однако, заранее предупредить всех, у кого возникнет такое желание, что процессор i386 (отчасти «благодаря» тяжелому наследию 8086) имеет одну из самых хаотичных и нелогичных систем команд в мире; особенно это становится заметно, как только мы покидаем уютный мир ограниченного режима и «плоской» модели памяти, в котором нас заботливо устроила операционная система, и встречаемся лицом к лицу с программированием дескрипторов сегментов, нелепыми прыжками между кольцами защиты и прочими «прелестями» платформы, с которыми приходится бороться создателям современных операционных систем.

Так что, если вас всерьёз заинтересовало низкоуровневое программирование, мы можем посоветовать поизучать другие архитектуры, например, процессоры SPARC. Впрочем, любопытство в любом случае не порок, и если вы готовы к определённым трудностям — то найдите любой справочник по i386 и изучайте на здоровье :-)

Глава 3. Ассемблер NASM

Ранее мы использовали ассемблер NASM, ограничиваясь лишь общими замечаниями и изредка отвлекаясь, чтобы описать некоторые его возможности, без которых не могли обойтись. Так, в § 1.5 было дано ровно столько пояснений, чтобы можно было понять одну простейшую программу. В § 2.2 нам потребовалось использовать память для хранения данных, и пришлось посвятить § 2.2.2 директивам резервирования памяти и меткам. Прежде чем привести в § 2.6.9 пример сложной подпрограммы, мы вынуждены были в § 2.6.8 рассказать про локальные метки.

Эту главу мы целиком посвятим изучению ассемблера NASM, начав с более формального, чем раньше, описания синтаксиса его языка. После этого мы изучим возможности его макропроцессора и закончим это всё кратким описанием ключей командной строки, используемых при запуске NASM.

§ 3.1. Синтаксис языка ассемблера NASM

Основной синтаксической единицей практически любого языка ассемблера (и NASM тут не исключение) является *строка текста*. Этим языки ассемблера отличаются от большинства (хотя и далеко не всех) языков высокого уровня, в которых символ перевода строки приравнивается к обычному пробелу.

Если по каким-либо причинам нам не хватило длины строки, чтобы уместить всё, что мы хотели в ней уместить, то можно воспользоваться средством «склеивания» строк. Если последним символом строки поставить «обратный слэш» (символ «\»), то ассемблер будет считать следующую строку продолжением предыдущей. Отметим, что это гораздо лучше, чем допускать в тексте программы очень длинные строки; обычно строка программы (любой, не только на языке ассемблера) не должна превышать 75 символов, хотя компиляторы этого от нас и не требуют.

Строка текста[1] на языке ассемблера NASM состоит (в общем случае) из четырёх полей: метки, имени команды, операндов и комментария, причём метка, имя команды и комментарий являются полями необязательными, что касается операндов, то требования к ним налагаются командой; если имя команды отсутствует, то отсутствуют и операнды. Могут отсутствовать и все четыре поля, тогда строка оказывается пустой. Ассемблер пустые строки игнорирует, но мы можем использовать их, чтобы визуально разделять между собой части программы.

В качестве метки можно использовать слово, состоящее из латинских букв, цифр, а также символов `'_'`, `'$'`, `'#'`, `'@'`, `'~'`, `'.'` и `'?'`, причём начинаться метка может только с буквы или символов `'_'`, `'?'` и `'.'`; как мы видели в § 2.6.8, метки, начинающиеся с точки, считаются *локальными*. Кроме того, в некоторых случаях имя метки можно предварить символом `'$'`; обычно это используется, если нужно создать метку, имя которой совпадает с именем регистра, команды или директивы[2]. Надо отметить, что ассемблер различает регистр букв в именах меток, то есть, например, `'label'`, `'LABEL'`, `'Label'` и `'LaBeL'` — это четыре разные метки. После метки, если она в строке присутствует, можно поставить символ двоеточия, но не обязательно. Как уже отмечалось, обычно программисты ставят двоеточия после меток, на которые можно передавать управление, и не ставят двоеточия после меток, обозначающих области памяти. Хотя ассемблер и не требует поступать именно так, программа при использовании этого соглашения становится яснее.

В поле имени команды, если оно присутствует, может быть обозначение машинной команды (возможно, с префиксом `rep`, см. стр. 91; существуют и другие префиксы), либо псевдокоманды — директивы специального вида (некоторые из них мы уже рассматривали, и к этому вопросу ещё вернёмся), либо, наконец, имя макроса (с такими мы тоже встречались, к ним относится, например, использовавшийся в примерах `PRINT`; созданию макросов будет посвящён отдельный параграф). В отличие от меток, в именах машинных команд и псевдокоманд ассемблер регистры букв не различает, так что мы можем с равным успехом написать, например, `mov`, `MOV`, `Mov` и даже `mOv`, хотя так писать, конечно же, не стоит. В именах макросов, как и в именах меток, регистр различается.

Требования к содержимому поля операндов зависят от того, какая конкретно команда, псевдокоманда или макрос указаны в поле команды. Если операндов больше одного, то они разделяются запятой. В поле

[1]Здесь и далее под «строкой» понимается в том числе и «логическая» строка, склеенная из нескольких строк с помощью обратных слэшей; в дальнейшем мы не будем уточнять, что имеем в виду именно такие строки.

[2]Такое может понадобиться только в случае, если ваша программа состоит из модулей, написанных на разных языках программирования; тогда в других модулях вполне могут встретиться метки, совпадающие по имени с ключевыми словами ассемблера, и может потребоваться возможность на них ссылаться.

операндов часто приходится использовать названия регистров, и в этих названиях регистр букв не различается, как и в именах машинных команд.

Читателю, запутавшемуся в том, где же регистр важен, а где нет, можно порекомендовать одно простое правило: **ассемблер nasm не различает заглавные и строчные буквы во всех словах, которые он ввёл сам: в именах команд, названиях регистров, директивах, псевдокомандах, обозначениях длины операндов и типа переходов (слова `byte`, `dword`, `near` и т. п.), но при этом считает заглавные и строчные разными буквами в тех именах, которые вводит пользователь (программист, пишущий на языке ассемблера) — в метках и именах макросов**.

Отметим ещё одно свойство NASM, связанное с записью операндов. **Операнд типа «память» *всегда* записывается с использованием квадратных скобок**. Для некоторых других ассемблеров это не так, что порождает постоянную путаницу.

Комментарий обозначается символом «точка с запятой» («`;`»). Начиная с этого символа, весь текст до конца строки ассемблер не принимает во внимание, что позволяет написать там всё, что угодно. Обычно это используют для вставки в текст программы пояснений, предназначенных для тех, кому придётся этот текст прочитать.

§ 3.2. Псевдокоманды

Под ***псевдокомандами*** понимается ряд вводимых ассемблером NASM слов, которые могут использоваться синтаксически так же, как и мнемоники машинных команд, хотя машинными командами на самом деле не являются. Некоторые такие псевдокоманды, а именно `db`, `dw`, `dd`, `resb`, `resw` и `resd` нам уже известны из § 2.2.2. Отметим только, что кроме перечисленных, NASM поддерживает также псевдокоманды `resq`, `rest`, `dq` и `dt`. Буква `q` в их названиях означает «quadro» — «учетверённое слово» (8 байт), буква `t` — от слова «ten» и означает десятибайтные элементы. Эти псевдокоманды могут потребоваться только в программе, работающей с числами с плавающей точкой (попросту говоря, дробными числами); более того, `dq` и `dt` в качестве инициализаторов допускают только, и исключительно, числа с плавающей точкой (например, `71.361775`). Кроме псевдокоманд `dq` и `dt`, числа с плавающей точкой принимает и псевдокоманда `dd`; это обусловлено тем, что стандарт IEEE-754[3] преду-

[3]IEEE-754 — это международный стандарт, описывающий способ представления в машинной памяти чисел с плавающей точкой и регламентирующий операции над ними; стандарт был создан под эгидой американской организации *Institute of Electrical and Electronics Engineers* (IEEE), представляющей собой профессиональную ассоциацию инженеров в области электротехники и электроники.

сматривает три формата чисел с плавающей точкой — обычные, двойной точности и повышенной точности, занимающие, соответственно, 4 байта, 8 байт и 10 байт.

Отдельного разговора заслуживает псевдокоманда `equ`, предназначенная для *определения констант*. Эта псевдокоманда всегда применяется в сочетании с меткой, то есть не поставить перед ней метку считается ошибкой. Псевдокоманда `equ` связывает стоящую перед ней метку с *явно заданным числом*. Самый простой пример:

```
four    equ 4
```

Здесь мы определили метку `four`, **задающую число 4**. Теперь, например,

```
        mov eax, four
```

есть то же самое, что и

```
        mov eax, 4
```

Уместно напомнить, что, вообще говоря, *любая метка представляет собой не более чем число*, но метки, введённые другим способом (помечающие другие строки) связываются с ***адресами в памяти*** (которые, разумеется, есть тоже ни что иное как просто числа).

Одно из самых частых применений директивы `equ` — это связать с некоторым именем (меткой) длину массива, только что заданного с помощью директивы `db`, `dw` или любой другой. Для этого используется ***псевдометка*** `$`, которая в каждой строчке, где она появляется, обозначает *текущий адрес*[4]. Например, можно написать так:

```
msg     db "Hello and welcome", 10, 0
msglen  equ $-msg
```

Выражение `$-msg`, представляющее собой разность двух чисел, известных ассемблеру во время его работы, будет просто вычислено прямо во время ассемблирования. Поскольку `$` означает адрес, ставший текущим уже *после* описания строки, а `msg` — адрес *начала* строки, то их разность в точности равна длине строки (в нашем примере 19). К вычислению выражений во время ассемблирования мы вернёмся в § 3.4.

Директива `times` позволяет повторить какую-нибудь команду (или псевдокоманду) заданное количество раз. Например,

```
stars  times 4096 db '*'
```

[4]Точнее говоря, текущее смещение относительно начала секции.

задаёт область памяти размером в 4096 байт, заполненную кодом символа `'*'`, точно так же, как это сделали бы 4096 одинаковых строк, содержащих директиву `db '*'`.

Иногда может оказаться полезной псевдокоманда `incbin`, позволяющая создать область памяти, заполненную данными из некоторого внешнего файла. Подробно мы её рассматривать не будем; заинтересованный читатель может изучить эту директиву самостоятельно, обратившись к документации.

§ 3.3. Константы

Константы в языке ассемблера NASM делятся на четыре категории: целые числа, символьные константы, строковые константы и числа с плавающей точкой.

Как уже говорилось (см. стр. 41), ***целочисленные константы*** можно задавать в десятичной, двоичной, шестнадцатеричной и восьмеричной системах счисления. Если просто написать число, состоящее из цифр (и, возможно, знака «минус» в качестве первого символа), то это число будет воспринято ассемблером как десятичное. Шестнадцатеричное число можно задать тремя способами: прибавив в конце числа букву `h` (например, `2af3h`), либо написав перед числом символ `$`, как в Borland Pascal (например, `$2af3`), либо поставив, опять таки, перед числом символы `0x`, как в языке Си (`0x2af3`). При использовании символа `$` необходимо следить, чтобы сразу после `$` стояла цифра, а не буква, так что если число начинается с буквы, необходимо добавить `0` (например, `$0f9` вместо просто `$f9`). Это необходимо, чтобы ассемблер не путал запись числа с записью пользовательской метки, перед которыми, как мы уже говорили, иногда тоже ставится знак `$`. Восьмеричное число обозначается добавлением после числа буквы `o` или `q` (например, `634o`, `754q`). Наконец, двоичное число обозначается буквой `b` (`10011011b`).

Символьные константы и ***строковые константы*** очень похожи друг на друга, и, более того, в любом месте, где по смыслу должна быть строковая константа, можно употребить и символьную. Разница между строковыми и символьными константами заключается только в их длине: под *символьной* константой подразумевается такая константа, которая укладывается в длину «двойного слова» (то есть содержит не более 4 символов) и может, в силу этого, рассматриваться как альтернативная запись целого числа (либо битовой строки). И символьные, и строковые константы могут записываться как с помощью двойных кавычек, так и с помощью апострофов. Это позволяет использовать в строках и сами символы апострофов и кавычек: если строка содержит символ кавычек одного типа, то её заключают в кавычки другого типа (см. пример на стр. 43).

Символьные константы, содержащие меньше 4 символов, считаются синонимами целых чисел, младшие байты которых равны кодам символов из константы, а недостающие старшие байты заполнены нулями. При использовании символьных констант следует помнить, что целые числа в компьютерах с процессорами i386 записываются в обратном порядке байтов, то есть младший байт идёт первым. В то же время, по смыслу строки (и символьной константы) код первой буквы должен в памяти размещаться первым. Поэтому, например, константа 'abcd' эквивалентна числу 64636261h: 64h — это код буквы d, 61h — код буквы a, и в обоих случаях байт со значением 61h стоит первым, а 64h — последним. В некоторых случаях ассемблер воспринимает в качестве строковых и такие константы, которые достаточно коротки и могли бы считаться символьными. Это происходит, например, если ассемблер видит символьную константу длиной более 1 символа в параметрах директивы db или константу длиной более двух символов в параметрах директивы dw.

Константы с плавающей точкой, задающие дробные числа, синтаксически отличаются от целочисленных констант наличием десятичной точки. Учтите, что **целочисленная константа 1 и константа 1.0 не имеют между собой ничего общего!** Для наглядности отметим, что битовая запись числа с плавающей точкой 1.0 одинарной точности (то есть запись, занимающая 4 байта, так же, как и для целого числа) эквивалентна записи целого числа 3f800000h (1065353216 в десятичной записи). Константу с плавающей точкой можно задать и в «экспоненциальном» виде, используя букву e или E. Например, 1.0e-5 есть то же самое, что и 0.00001. Обратите внимание, что десятичная точка по-прежнему обязательна.

§ 3.4. Вычисление выражений во время ассемблирования

Ассемблер NASM в некоторых случаях вычисляет встретившиеся ему арифметические выражения непосредственно во время ассемблирования. Важно понимать, что **в итоговый машинный код попадают только вычисленные результаты, а не сами действия по их вычислению**. Естественно, для вычисления выражения во время ассемблирования необходимо, чтобы такое выражение не содержало никаких неизвестных: всё, что нужно для вычисления, должно быть известно ассемблеру во время его работы.

§ 3.4.1. Вычисляемые выражения и операции в них

Выражение, вычисляемое ассемблером, должно быть **целочисленным**, то есть состоять из целочисленных констант и меток, и использовать операции из следующего списка:

- + и - — сложение и вычитание
- * — умножение;
- / и % — целочисленное деление и остаток от деления (для беззнаковых целых чисел);
- // и %% — целочисленное деление и остаток от деления (для знаковых целых чисел);
- &, |, ^ — операции побитового «и», «или», «исключающего или»;
- << и >> — операции побитового сдвига влево и вправо;
- унарные операции - и + используются в их обычной роли: - меняет знак числа на противоположный, + не делает ничего;
- унарная операция ~ обозначает побитовое отрицание.

При применении операций % и %% необходимо обязательно оставлять пробельный символ после знака операции, чтобы ассемблер не перепутал их с *макродирективами* (макродирективы мы рассмотрим позже).

Ещё одна унарная операция, seg, для нас неприменима ввиду отсутствия сегментов в «плоской» модели памяти.

Унарные операции имеют самый высокий приоритет, следом за ними идут операции умножения, деления и остатка от деления, ещё ниже приоритет у операций сложения и вычитания. Далее (в порядке убывания приоритета) идут операции сдвигов, операция &, затем операция ^, и замыкает список операция |, имеющая самый низкий приоритет. Порядок выполнения операций можно изменить, применив круглые скобки.

§ 3.4.2. Критические выражения

Ассемблер анализирует исходный текст в два прохода. На первом проходе вычисляется размер всех машинных команд и других данных, подлежащих размещению в памяти программы; в результате этого ассемблер устанавливает, какое *числовое значение* должно быть приписано каждой из встретившихся в тексте программы меток. На втором проходе генерируется собственно машинный код и прочее содержимое памяти. Второй проход нужен, чтобы, например, можно было ссылаться на метку, стоящую в тексте *позже*, чем ссылка на неё: когда ассемблер видит метку, скажем, в команде jmp, раньше, чем встретится собственно команда, помеченная этой меткой, на первом проходе он не может сгенерировать код, поскольку не знает численного значения метки. На втором проходе все значения уже известны, и никаких проблем с генерированием кода не возникает.

Всё это имеет прямое отношение к механизму вычисления выражений. Ясно, что выражение, содержащее метку, ассемблер может вычислить на первом проходе только в случае, если метка стояла в тексте раньше, чем вычисляемое выражение; в противном случае вычисление выражения приходится отложить до второго прохода. Ничего страшного в этом нет, **если только значение выражения не влияет на размер команды, выделяемой области памяти и т. п.**, то есть от значения этого выражения не зависят численные значения, которые нужно будет приписать дальнейшим встреченным меткам. Если же это условие не выполнено, то невозможность вычислить выражение на первом проходе приведёт к невозможности выполнить задачу первого прохода — определить численные значения всех меток. Более того, в некоторых случаях не помогло бы никакое количество проходов, даже если бы ассемблер это умел. В документации к ассемблеру NASM приведён такой пример:

```
        times (label-$) db 0
label:  db      'Where am I?'
```

Здесь строчка с директивой `times` должна ввести столько нулевых байтов, насколько метка `label` отстоит от самой этой строчки — но ведь метка `label` как раз и отстоит от этой строчки настолько, сколько нулевых байтов будет введено. Так сколько же их должно быть введено?!

В связи с этим мы вводим понятие ***критического выражения***: это такое выражение, вычисляемое во время ассемблирования, которое ассемблеру необходимо вычислить во время первого прохода. Критическими ассемблер считает любые выражения, от которых тем или иным образом зависит размер чего бы то ни было, располагаемого в памяти (и которые, следовательно, могут повлиять на значения меток, вводимых позже). В критических выражениях можно использовать только числовые константы, а также метки, определённые *выше по тексту программы*, чем рассматриваемое выражение. Это гарантирует возможность вычисления выражения на первом проходе.

Кроме аргумента директивы `times`, к категории критических относятся, например, выражения в аргументах псевдокоманд `resb`, `resw` и др., а также в некоторых случаях — выражения в составе исполнительных адресов, которые могут повлиять на итоговый размер ассемблируемой команды. Так, команды «`mov eax,[ebx]`», «`mov eax,[ebx+10]`» и «`mov eax,[ebx+10000]`» порождают соответственно 2 байта, 3 байта и 6 байтов кода, поскольку число, входящее в состав исполнительного адреса, в первом случае занимает всего 1 байт, во втором — 2, а в последнем — 4; но сколько памяти займёт команда

```
        mov eax, [ebx+label]
```

если значение label пока не определено? Впрочем, этих трудностей можно избежать, если внутри исполнительного адреса в явном виде указать разрядность словом byte, word или dword. Так, если написать

```
mov eax, [ebx + dword label]
```

то, даже если значение label ещё не известно, длина его (и, как следствие, длина всей машинной команды) уже указана.

§ 3.4.3. Выражения в составе исполнительного адреса

На рис. 2.2 (см. стр. 49) мы приводили общий вид исполнительного адреса (операндов типа «память») с точки зрения машинных команд. Ассемблер NASM может воспринимать и более сложные выражения в квадратных скобках, лишь бы их было возможно привести к указанному виду. Так, например, в команде

```
mov eax, [5*ebx]
```

используется умножение на число 5, что вроде бы запрещено (умножать можно только на 1, 2, 4 и 8), но ассемблер справляется с этой сложностью, приведя в команде операнд к виду [ebx+4*ebx], который уже вполне корректен. Если же рассмотреть команду

```
mov eax, [ebx+4*ecx+5*x+y]
```

в которой x и y — некоторые метки, то и с этим ассемблер справится, попросту *вычислив* выражение 5*x+y и получив в итоге одно число, что уже вполне соответствует общему виду исполнительного адреса.

Необходимо только помнить, что, если только в явном виде не указать нужную разрядность, такие выражения будут считаться *критическими*, то есть должны зависеть только от меток, уже введённых к моменту рассмотрения выражения (см. предыдущий параграф).

§ 3.5. Макросредства и макропроцессор

§ 3.5.1. Основные понятия

Под ***макропроцессором*** понимают программное средство, которое получает на вход некоторый текст и, пользуясь указаниями, данными в самом тексте, частично преобразует его, давая на выходе, в свою очередь, текст, но уже не имеющий указаний к преобразованию. В применении к языкам программирования макропроцессор — это преобразователь исходного текста программы, обычно совмещённый с компилятором;

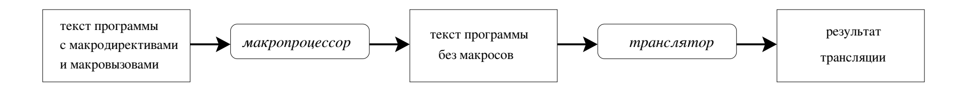


Рис. 3.1. Схема работы макропроцессора

результатом работы макропроцессора является **текст на языке программирования**, который потом уже обрабатывается компилятором в соответствии с правилами языка (см. рис. 3.1).

Поскольку языки ассемблера обычно весьма бедны по своим изобразительным возможностям (если сравнивать их с языками высокого уровня), то, чтобы хоть как-то компенсировать программистам неудобства, обычно ассемблеры снабжают очень мощными макропроцессорами. В частности, рассматриваемый нами ассемблер NASM содержит в себе алгоритмически полный макропроцессор, который мы можем при желании заставить написать за нас едва ли не всю программу.

С макросами мы уже встречались: часто использовавшиеся нами `PRINT` и `FINISH` представляют собой именно макросы, или, точнее, ***имена макросов***.

Вообще, ***макросом*** называют некоторое правило, в соответствии с которым фрагмент программы, содержащий определённое слово, должен быть преобразован. Само это слово называют ***именем макроса***; часто вместо термина «имя макроса» используют просто слово «макрос», хотя это и не совсем верно.

Прежде чем мы сможем воспользоваться макросом, его необходимо определить, то есть, во-первых, указать макропроцессору, что некий идентификатор отныне считается именем макроса (так что его появление в тексте программы требует вмешательства макропроцессора), и, во-вторых, задать то правило, по которому макропроцессор должен действовать, встретив это имя. Фрагмент программы, определяющий макрос, называют ***макроопределением***. Когда макропроцессор встречает в тексте программы имя макроса и параметры (так называемый ***вызов макроса***, или ***макровызов***), он *заменяет* имя макроса (и, возможно, параметры, относящиеся к нему) неким фрагментом текста, полученным в соответствии с определением макроса. Такая замена называется ***макроподстановкой***, а текст, полученный в результате — *макрорасширением*[5].

Бывает и так, что макропроцессор производит преобразование текста программы, не видя ни одного имени макроса, но повинуясь ещё более прямым указаниям, выраженным в виде ***макродиректив***. Одну такую макродирективу мы уже знаем: это директива `%include`, которая

[5]Термин «макрорасширение» — это не слишком удачная калька с соответствующего английского термина «macro expansion».

приказывает макропроцессору заменить её саму на содержимое файла, указанного параметром директивы. Так, привычная нам строка

```
%include "stud_io.inc"
```

заменяется на всё, что есть в файле stud_io.inc.

§ 3.5.2. Простейшие примеры макросов

Чтобы составить представление о том, как можно воспользоваться макропроцессором и для чего он нужен, приведём два простых примера. Как мы видели из §§ 2.6.6, 2.6.7 и 2.6.9, запись вызова подпрограммы на языке ассемблера занимает несколько строк (если быть точными, $2 + n$, где n — число параметров подпрограммы). Это не всегда удобно, особенно для людей, привыкших к языкам высокого уровня. Пользуясь механизмом макросов, мы можем изрядно сократить запись вызова подпрограммы. Для этого мы опишем макросы pcall1, pcall2 и т. д., для вызова, соответственно, процедуры от одного параметра, двух параметров и т. д. С помощью таких макросов запись вызова процедуры сократится до одной строчки; например, вместо

```
        push edx
        push dword mylabel
        push dword 517
        call myproc
        add esp, 12
```

можно будет написать

```
        pcall3 myproc, dword 517, dword mylabel, edx
```

что, конечно, гораздо удобнее и понятнее. Позже, разобравшись с макроопределениями глубже, мы перепишем эти макросы, вместо них введя один макрос pcall, работающий для любого количества аргументов, но пока для примера ограничимся частными случаями. Итак, пишем макроопределение:

```
%macro pcall1 2       ; 2 -- кол-во параметров макроса
        push %2
        call %1
        add esp, 4
%endmacro
```

Мы описали ***многострочный макрос*** с именем pcall1, имеющий два параметра: имя вызываемой процедуры для команды call и аргумент

процедуры для занесения в стек. Строки, написанные между директивами `%macro` и `%endmacro`, составляют ***тело макроса*** — шаблон для текста, который должен получиться в результате макроподстановки. Сама макроподстановка в данном случае будет довольно простой: макропроцессор только заменит вхождения `%1` и `%2` соответственно на первый и второй параметры, заданные в макровызове. Если после такого определения в тексте нашей программы встретится строка вида

```
        pcall1 proc, eax
```

макропроцессор воспримет эту строку как макровызов и выполнит макроподстановку в соответствии с вышеприведённым макроопределением, считая первым параметром слово `proc`, вторым параметром слово `eax` и подставляя их вместо `%1` и `%2`. В результате получится следующий фрагмент:

```
        push eax
        call proc
        add esp, 4
```

Аналогичным образом опишем макросы `pcall2` и `pcall3`:

```
%macro pcall2 3
        push %3
        push %2
        call %1
        add esp, 8
%endmacro
%macro pcall3 4
        push %4
        push %3
        push %2
        call %1
        add esp, 12
%endmacro
```

Для полноты можно дописать также и макрос `pcall0`:

```
%macro pcall0 1
        call %1
%endmacro
```

Конечно, такой макрос, в отличие от предыдущих, ничуть не сокращает объём программы, но зато он позволит нам все вызовы подпрограмм оформить единообразно. Описание макросов `pcall4`, `pcall5` и т. д. до

pcall8 оставляем читателю в качестве упражнения; заодно для самопроверки ответьте на вопрос, почему мы предлагаем остановиться именно на pcall8, а не, например, на pcall9 или pcall12.

Рассмотренный нами пример использовал *многострочный макрос*; как мы убедились, вызов многострочного макроса синтаксически выглядит точно так же, как использование машинных команд или псевдокоманд: вместо имени команды пишется имя макроса, затем через запятую перечисляются параметры. При этом многострочный макрос всегда преобразуется в одну или несколько *строк* на языке ассемблера. Но что если, к примеру, нам нужно сгенерировать с помощью макроса некоторую *часть строки*, а не фрагмент из нескольких строк? Такая потребность тоже возникает довольно регулярно. Так, в примере, приведённом в § 2.6.9, видно, что внутри процедур очень часто приходится использовать конструкции вроде [ebp+12], [ebp-4] и т. п. для обращения к параметрам процедуры и её локальным переменным. В принципе, к этим конструкциям несложно привыкнуть; но можно пойти и другим путём, применив ***однострочные макросы***. Для начала напишем следующие[6] макроопределения:

```
%define arg1 ebp+8
%define arg2 ebp+12
%define arg3 ebp+16
%define local1 ebp-4
%define local2 ebp-8
%define local3 ebp-12
```

В дополнение к ним допишем ещё и такое:

```
%define arg(n) ebp+(4*n)+4
%define local(n) ebp-(4*n)
```

Теперь к параметру процедуры можно обратиться так:

```
        mov eax, [arg1]
```

или так (если, например, не хватило описанных макросов)

```
        mov [arg(7)], edx
```

В принципе мы могли и квадратные скобки включить внутрь макросов, чтобы не писать их каждый раз. Например, если изменить определение макроса arg1 на следующее:

[6]Здесь и далее в наших примерах мы предполагаем, что все параметры процедур и все локальные переменные всегда представляют собой «двойные слова», то есть имеют размер 4 байта; на самом деле, конечно, это не всегда так, но нам сейчас важнее иллюстративная ценность примера.

```
%define arg1 [ebp+8]
```

то соответствующий макровызов стал бы выглядеть так:

```
mov eax, arg1
```

Мы не сделали этого из соображений сохранения наглядности. Ассемблер NASM поддерживает, как мы уже знаем, соглашение о том, что любое обращение к памяти оформляется с помощью квадратных скобок, если же их нет, то мы имеем дело с непосредственным или регистровым операндом. Программист, привыкший к этому соглашению, при чтении программы будет вынужден прилагать лишние усилия, чтобы вспомнить, что `arg1` в данном случае не метка, а имя макроса, так что здесь происходит именно обращение к памяти, а не загрузка в регистр адреса метки. Понятности программы такие вещи отнюдь не способствуют. Учтите, что и вы сами, будучи даже автором программы, можете за несколько дней начисто забыть, что́ же имелось в виду, и тогда экономия двух символов (скобок) обернётся для вас потерей бесценного времени.

§ 3.5.3. Однострочные макросы; макропеременные

Как видно из примеров предыдущего параграфа, однострочный макрос — это такой макрос, определение которого состоит из одной строки, а его вызов разворачивается во фрагмент строки текста (то есть может использоваться для генерации *части* строки). Отметим, что единожды определённый макрос можно при необходимости *переопределить*, просто вставив в текст программы ещё одно определение того же самого макроса. С того момента, как макропроцессор «увидит» новое определение, он будет использовать его вместо старого. Таким образом, одно и то же имя макроса в разных местах программы может означать разные вещи и раскрываться в разные фрагменты текста. Более того, макрос вообще можно убрать, воспользовавшись директивой `%undef`; встретив такую директиву, макропроцессор немедленно «забудет» о существовании макроса. Представляет интерес вопрос о том, что будет, если в определении одного макроса использовать вызов другого макроса, а этот последний, в свою очередь, время от времени переопределять.

Если для описания однострочного макроса `A` использовать уже знакомую нам директиву `%define` и в её теле использовать макровызов макроса `B`, то этот макровызов в самой директиве не раскрывается; макропроцессор оставляет вхождение макроса `B` как оно есть до тех пор, пока не встретит вызов макроса `A`. Когда же будет выполнена макроподстановка для `A`, в её результате будет содержаться `B`, и для него макропроцессор, в свою очередь, выполнит макроподстановку. Таким образом, будет использовано то определение макроса `B`, которое было актуальным в момент подстановки `A`.

Поясним сказанное на примере. Пусть мы ввели два макроса:

```
%define         thenumber       25
%define         mkvar           dd thenumber
```

Если теперь написать в программе строчку

```
var1            mkvar
```

то макропроцессор сначала выполнит макроподстановку для `mkvar`, получив строку

```
var1            dd thenumber
```

а из неё, в свою очередь, макроподстановкой `thenumber` получит строку

```
var1            dd 25
```

Если теперь переопределить `thenumber` и снова воспользоваться вызовом `mkvar`:

```
%define         thenumber       36
var2            mkvar
```

то результатом работы макропроцессора будет строка, содержащая именно число 36:

```
var2            dd 36
```

несмотря на то, что сам макрос `mkvar` мы не изменяли: на первом шаге будет получено, как и в прошлый раз, `dd thenumber`, но у `thenumber` теперь значение 36, оно и будет подставлено. Такая стратегия макроподстановок называется «ленивой[7]». Однако ассембер NASM позволяет применять и другую стратегию, называемую *энергичной*, для чего предусмотрена директива `%xdefine`. Эта директива полностью аналогична директиве `%define` с той только разницей, что, если в теле описания макроса встречаются макровызовы, макропроцессор производит их макроподстановки незамедлительно, не дожидаясь, пока пользователь вызовет описываемый макрос. Так, если в вышеприведённом примере заменить директиву `%define` в описании макроса `mkvar` на `%xdefine`:

```
%define         thenumber       25
%xdefine        mkvar           dd thenumber
var1            mkvar
%define         thenumber       36
var2            mkvar
```

[7]Такое название является калькой английского *lazy* и частично оправдано тем, что макропроцессор как бы «ленится» выполнять макроподстановку (в данном случае макроса `thenumber`), пока его к этому не вынудят.

то обе получившиеся строки будут содержать число 25:

```
var1            dd 25
var2            dd 25
```

Переопределение макроса thenumber теперь не в силах повлиять на работу макроса mkvar, поскольку тело макроса mkvar на этот раз не содержит слова thenumber: обрабатывая определение mkvar, макропроцессор подставил вместо слова thenumber его значение (25).

Иногда возникает потребность связать с именем макроса не просто строку, а число, являющееся результатом вычисления арифметического выражения. Ассемблер NASM позволяет это сделать, используя директиву %assign. В отличие от %define и %xdefine, эта директива не только выполняет все необходимые подстановки в теле макроопределения, но и пытается *вычислить* тело как обыкновенное целочисленное арифметическое выражение. Если это не получается, фиксируется ошибка. Так, если написать в программе сначала

```
%assign         var     25
```

а потом

```
%assign         var     var+1
```

то в результате с макроименем var будет связано значение 26, которое и будет подставлено, если макропроцессор встретит слово var в дальнейшем тексте программы.

Макроимена, вводимые директивой %assign, обычно называют ***макропеременными***. Как мы увидим далее, макропеременные являются важным средством, позволяющим задать макропроцессору целую программу, результатом которой может стать очень длинный текст на языке ассемблера.

§ 3.5.4. Условная компиляция

Часто при разработке программ возникает потребность в создании различных версий исполняемого файла с использованием одного и того же исходного текста. Допустим, мы пишем программы на заказ и у нас есть два заказчика Петров и Сидоров, причём программы для них почти одинаковы, но у каждого из двоих имеются специфические потребности, отсутствующие у другого. В такой ситуации хотелось бы, конечно, иметь и поддерживать один исходный текст: в противном случе у нас появятся две копии одного и того же кода, и придётся, например, каждую найденную ошибку исправлять в двух местах. Однако при компиляции версии

для Петрова нужно исключить из работы фрагменты, предназначенные для Сидорова, и наоборот.

Подобная потребность возникает и в других ситуациях. Известно, например, что отладочная печать (то есть вставка в программу специальных операций вывода, позволяющих понять, что происходит во время работы программы) является одним из самых универсальных и мощных средств отладки программ; в то же время окончательная версия программы, разумеется, не должна содержать операций отладочной печати, поскольку вся отладочная информация предназначена для программиста, автора программы, а пользователю может только мешать. Проблема в том, что отладка программы — процесс бесконечный, и как только мы решим, что она завершена и удалим из текста всю отладочную печать, по закону подлости тут же обнаружится очередная ошибка, и нам вновь придётся редактировать наш исходник, чтобы вернуть отладочную печать на место.

Большинство профессиональных компилируемых языков программирования поддерживают для подобных случаев специальные конструкции, называемые ***директивами условной компиляции*** и позволяющие выбирать, какие фрагменты программы компилировать, а какие игнорировать. Обычно отработку директив условной компиляции возлагают на макропроцессор, если, конечно, таковой в языке предусмотрен. Сказанное справедливо, кроме прочего, практически для всех языков ассемблера, включая и наш NASM.

Рассмотрим пример, связанный с отладкой. Допустим, мы написали программу, откомпилировали её и запустили, но она завершается аварийно, и мы не можем понять причину, но думаем, что авария происходит в некоем «подозрительном» фрагменте. Чтобы проверить своё предположение, мы хотим непосредственно перед входом в этот фрагмент и сразу после выхода из него вставить печать соответствующих сообщений. Чтобы нам не пришлось по несколько раз стирать эти сообщения и вставлять их снова, воспользуемся директивами условной компиляции. Выглядеть это будет примерно так:

```
%ifdef DEBUG_PRINT
        PRINT "Entering suspicious section"
        PUTCHAR 10
%endif
;
;     здесь идёт "подозрительная" часть программы
;
%ifdef DEBUG_PRINT
        PRINT "Leaving suspicious section"
        PUTCHAR 10
```

```
%endif
```

Здесь %ifdef — это одна из ***директив условной компиляции***, означающая «компилировать только в случае, если определён данный однострочный макрос» (в данном случае это макрос DEBUG_PRINT). Теперь в начало программы следует вставить строку, определяющую этот символ:

```
%define DEBUG_PRINT
```

Тогда при запуске NASM «увидит» и откомпилирует фрагменты нашего исходного текста, заключённые между соответствующими %ifdef и %endif; когда же мы найдём ошибку и отладочная печать будет нам больше не нужна, достаточно будет убрать этот %define из начала программы или даже поставить перед ним знак комментария:

```
;%define DEBUG_PRINT
```

и фрагменты, обрамлённые соответствующими директивами, макропроцессор будет попросту игнорировать, так что их можно совершенно спокойно оставить в тексте программы, а не удалять, на случай, если они снова понадобятся.

Забегая вперёд, отметим, что для включения и отключения отладочной печати, оформленной таким образом, можно вообще обойтись без правки исходного текста. Определить макросимвол можно ключом командной строки NASM; в частности, включить отладочную печать из нашего примера можно, вызвав NASM примерно таким образом:

```
nasm -f elf -dDEBUG_PRINT prog.asm
```

что избавляет нас от необходимости вставлять в исходный текст директиву %define, а потом её удалять.

Возвращаясь к ситуации с двумя заказчиками, мы можем предусмотреть в программе конструкции, подобные следующей:

```
%ifdef FOR_PETROV
;
; здесь код, предназначенный только для Петрова
;
%elifdef FOR_SIDOROV
;
; а здесь - только для Сидорова
;
%else
; если ни тот символ, ни другой не определены,
; прервём компиляцию и выдадим сообщение об ошибке
%error Please define either FOR_PETROV or FOR_SIDOROV
%endif
```

(директива `%elifdef` — это сокращённая форма записи для `else` и `ifdef`). При компиляции такой программы нужно будет обязательно указать ключ `-dFOR_PETROV` или `-dFOR_SIDOROV`, иначе NASM начнёт обрабатывать фрагмент, находящийся после `%else`, и, встретив директиву `%error`, выдаст сообщение об ошибке.

Кроме проверки *наличия* макросимвола, можно проверять также и факт *отсутствия* макросимвола (то есть прямо противоположное условие). Это делается директивой `%ifndef` (*if not defined*). Как и для `%ifdef`, для `%ifndef` существует сокращённая запись конструкции с `%else`, она называется `%elifndef`.

Для задания условия, при котором тот или иной фрагмент подлежит или не подлежит компиляции, можно пользоваться не только фактом наличия или отсутствия макроса; NASM поддерживает и другие директивы условной компиляции. Наиболее общей является директива `%if`, в которой условие задаётся арифметико-логическим выражением, вычисляемым во время компиляции. С такими выражениями мы уже встречались в § 3.4.1; для формирования логических выражений набор допустимых операций расширяется операциями =, <, >, >=, <=, в их обычном смысле, операцию «не равно» можно задать символом <>, как в Паскале, или символом `!=`, как в Си; поддерживается и Си-подобная форма записи операции «равно» в виде двух знаков равенства ==. Кроме того, доступны логические связки `&&` («и»), `||` («или») и `^^` («исключающее или»). Отметим, что все выражения, используемые в директиве `%if`, рассматриваются как *критические* (см. § 3.4.2). Так же, как и для всех остальных `%if`-директив, для простого `%if` имеется форма сокращённой записи конструкции с `%else` — директива `%elif`.

Перечислим кратко остальные поддерживаемые NASM условные директивы. Директивы `%ifidn` и `%ifidni` принимают два аргумента, разделённые запятой, и сравнивают их как строки, предварительно произведя, если это необходимо, макроподстановки в тексте аргументов. Фрагмент кода, следующий за этими директивами, транслируется только в случае, если строки окажутся равными, причём `%ifidn` требует точного совпадения, тогда как `%ifidni` игнорирует регистр и считает, например, строки `foobar`, `FooBar` и `FOOBAR` одинаковыми. Для проверки противоположного условия можно использовать директивы `%ifnidn` и `%ifnidni`; все четыре директивы имеют `%elif`-формы, соответственно, `%elifidn`, `%elifidni`, `%elifnidn` и `%elifnidni`. Директива `%ifmacro` проверяет существование многострочного макроса; поддерживаются директивы `%ifnmacro`, `%elifmacro` и `%elifnmacro`. Директивы `%ifid`, `%ifstr` и `%ifnum` проверяют, является ли их аргумент, соответственно, идентификатором, строкой или числовой константой. Как обычно, NASM поддерживает все дополнительные формы вида `%ifnXXX`, `%elifXXX` и `%elifnXXX` для всех трёх директив.

Кроме перечисленных, NASM поддерживает директиву `%ifctx` и соответствующие формы, но объяснение её работы достаточно сложно и обсуждать эту директиву мы не будем.

§ 3.5.5. Макроповторения

При необходимости препроцессор NASM можно заставить многократно (циклически) обрабатывать один и тот же фрагмент кода. Это достигается директивами `%rep` (от слова *repetition*) и `%endrep`. Директива `%rep` принимает один обязательный параметр, означающий количество повторений. Фрагмент кода, заключённый между директивами `%rep` и `%endrep`, будет обработан макропроцессором (и ассемблером) столько раз, сколько указано в параметре директивы `%rep`. Кроме того, между директивами `%rep` и `%endrep` может встретиться директива `%exitrep`, которая досрочно прекращает выполнение макроповторения. Рассмотрим простой пример. Пусть нам необходимо описать область памяти, состоящую из 100 последовательных байтов, причём в первом из них должно содержаться число 50, во втором — число 51 и т. д., в последнем, соответственно, число 149. Конечно, можно просто написать сто строк кода:

```
        db 50
        db 51
        db 52
        ;....
        db 148
        db 149
```

но это, во-первых, утомительно, а во-вторых, занимает слишком много места в тексте программы. Гораздо правильнее будет поручить генерацию этого кода макропроцессору, воспользовавшись макроповторением и макропеременной:

```
%assign n 50
%rep 100
        db n
%assign n n+1
%endrep
```

Встретив такой фрагмент, макропроцессор сначала свяжет с макропеременной `n` значение 50, затем сто раз рассмотрит две строчки, заключённые между `%rep` и `%endrep`, причём каждое рассмотрение этих строк приведёт к генерации очередной подлежащей ассемблированию строки `db 50`, `db 51`, `db 52` и т. д.; изменение числа происходит благодаря тому, что значение макропеременной `n` изменяется (увеличивается на единицу)

на каждом проходе макроповторения. Иначе говоря, в результате обработки макропроцессором этого фрагмента как раз и получатся точно такие сто строк кода, как показано выше, и именно они и будут ассемблироваться. Макропроцессор, таким образом, избавляет нас от необходимости писать эти сто строк вручную.

Рассмотрим более сложный пример. Пусть имеется необходимость задать область памяти, содержащую последовательно в виде четырёхбайтных целых все числа Фибоначчи[8], не превосходящие 100000. Сгенерировать соответствующую последовательность директив dd можно с помощью такого фрагмента кода:

```
fibonacci
        %assign i 1
        %assign j 1
        %rep 100000
          %if j > 100000
            %exitrep
          %endif

                dd j

          %assign k j+i
          %assign i j
          %assign j k
        %endrep
fib_count       equ ($-fibonacci)/4
```

причём метка fibonacci будет связана с адресом начала сгенерированной области памяти, а метка fib_count — с общим количеством чисел, размещённых в этой области памяти (с этим приёмом мы уже сталкивались на стр. 98).

Использовать макроповторения можно не только для генерации областей памяти, заполненных числами, но и для других целей. Пусть, например, у нас имеется массив из 128 двухбайтовых целых чисел:

```
array   resw 128
```

и мы хотим написать последовательность из 128 команд inc, увеличивающих на единицу каждый из элементов этого массива. Можно сделать это так:

```
%assign a array
```

[8]Напомним, что числа Фибоначчи — это последовательность чисел, начинающаяся с двух единиц, каждое следующее число которой получается сложением двух предыдущих: 1, 1, 2, 3, 5, 8, 13, 41, 34 и т. д.

```
%rep 128
        inc word [a]
%assign a a+2
%endrep
```

Читатель мог бы отметить, что использование в такой ситуации 128 команд нерационально и правильнее было бы воспользоваться циклом во время исполнения, например, так:

```
        mov ecx, 128
lp:     inc word [array + ecx*2 - 2]
        loop lp
```

В большинстве случаев такой вариант действительно предпочтительнее, поскольку такие три команды, естественно, будут занимать в несколько десятков раз меньше памяти, чем последовательность из 128 команд inc, но следует иметь в виду, что работать такой код будет примерно в полтора раза медленнее, так что в некоторых случаях применение макроцикла для генерации последовательности одинаковых команд (вместо цикла времени исполнения) может оказаться осмысленным.

§ 3.5.6. Многострочные макросы и локальные метки

Вернёмся теперь к многострочным макросам; такие макросы генерируют не фрагмент строки, а фрагмент текста, состоящий из нескольких строк. Описание многострочного макроса также состоит из нескольких строк, заключённых между директивами %macro и %endmacro. В § 3.5.2 мы уже рассматривали простейшие примеры многострочных макросов, однако в мало-мальски сложном случае рассмотренных средств нам не хватит. Пусть, например, мы хотим описать макрос zeromem, принимающий на вход два параметра — адрес и длину области памяти — и раскрывающийся в код, заполняющий эту память нулями. Не особенно задумываясь над происходящим, мы могли бы написать, например, следующий (**неправильный!**) код:

```
%macro zeromem 2 ; (два параметра - адрес и длина)
        push ecx
        push esi
        mov ecx, %2
        mov esi, %1
lp:     mov byte [esi], 0
        inc esi
        loop lp
        pop esi
        pop ecx
%endmacro
```

NASM примет такое описание и даже позволит произвести один макровызов. Если же в нашей программе встретятся хотя бы два вызова макроса `zeromem`, то при попытке оттранслировать программу мы получим сообщение об ошибке — NASM пожалуется на то, что мы используем одну и ту же метку (`lp:`) дважды. Действительно, при каждом макровызове макропроцессор вставит вместо вызова всё тело нашего макроопределения, только заменив `%1` и `%2` на соответствующие параметры, а всё остальное сохранив без изменения. Значит, строка

```
lp:     mov byte [esi], 0
```

содержащая метку `lp`, встретится ассемблеру (уже после макропроцессирования) дважды — или, точнее, ровно столько раз, сколько раз будет вызван макрос `zeromem`.

Ясно, что необходим некий механизм, позволяющий локализовать метку, используемую внутри многострочного макроса, с тем, чтобы такие метки, полученные вызовом одного и того же макроса в разных местах программы, не конфликтовали друг с другом. В NASM такой механизм называется «локальные метки в макросах». Чтобы задействовать этот механизм, необходимо начать имя метки с двух символов `%` — так, в приведённом выше примере оба вхождения метки `lp` нужно заменить на `%%lp`. Такая метка будет в каждом следующем макровызове заменяться некоторым новым (не повторяющимся) идентификатором. Отметим для наглядности, что при первом вызове макроса `zeromem` NASM заменит `%%lp` на `..@1.lp`, при втором — на `..@2.lp` и т. д.

Отметим ещё один недостаток вышеприведённого определения `zeromem`. Если при вызове этого макроса пользователь (программист, пользующийся нашим макросом, или, возможно, мы сами) использует в качестве первого параметра (адреса начала области памяти) регистр `ECX` или в качестве второго (длины области памяти) — регистр `ESI`, макровызов будет успешно оттранслирован, но работать программа будет совсем не так, как от неё ожидается. Действительно, если написать что-то вроде

```
section .bss
array   resb 256
arr_len equ $-array

section .text
; ...
        mov ecx, array
        mov esi, arr_len
        zeromem ecx, esi
; ...
```

то начало макроса `zeromem` развернётся в следующий код:

```
        push ecx
```

```
        push esi
        mov ecx, esi
        mov esi, ecx
        ; ...
```

в результате чего, очевидно, в **обоих** регистрах ECX и ESI окажется длина массива, а адрес его начала будет потерян. Скорее всего, программа в таком виде аварийно завершится, дойдя до этого фрагмента кода.

Чтобы избежать подобных проблем, можно воспользоваться директивами условной компиляции, проверяя, не является ли первый параметр регистром ECX и не является ли второй параметр регистром ESI, но можно поступить и проще — загрузить значения параметров в регистры через временную запись их в стек, то есть вместо

```
        mov ecx, %2
        mov esi, %1
```

написать

```
        push dword %2
        push dword %1
        pop esi
        pop ecx
```

Окончательно наше макроопределение примет следующий вид:

```
%macro zeromem 2 ; (два параметра - адрес и длина)
        push ecx
        push esi
        push dword %2
        push dword %1
        pop esi
        pop ecx
%%lp:   mov byte [esi], 0
        inc esi
        loop %%lp
        pop esi
        pop ecx
%endmacro
```

§ 3.5.7. Макросы с переменным числом параметров

При описании многострочных макросов с помощью директивы `%macro` ассемблер NASM позволяет задать переменное число параметров. Это делается с помощью символа `-`, который в данном случае символизирует тире. Например, директива

```
%macro mymacro 1-3
```

задаёт макрос, принимающий от одного до трёх параметров, а директива

```
%macro mysecondmacro 2-*
```

задаёт макрос, допускающий произвольное количество параметров, не меньшее двух. При работе с такими макросами может оказаться полезным обозначение %0, вместо которого макропроцессор во время макроподстановки подставляет число, равное фактическому количеству параметров.

Напомним, что сами аргументы многострочного макроса в его теле обозначаются как %1, %2 и т. д., но средств индексирования (то есть способа извлечь n-ый параметр, где n вычисляется уже во время макроподстановки) NASM не предусматривает. Как же в таком случае использовать параметры, если даже их количество заранее не известно? Проблему решает директива %rotate, позволяющая переобозначить параметры. Рассмотрим самый простой вариант директивы:

```
%rotate 1
```

Числовой параметр обозначает, на сколько позиций следует сдвинуть номера параметров. В данном случае это число 1, так что параметр, ранее обозначавшийся %2, после этой директивы будет иметь обозначение %1, в свою очередь бывший %3 превратится в %2 и т. д., ну а параметр, стоявший самым первым и имевший обозначение %1, в силу «цикличности» нашего сдвига получит номер, равный общему количеству параметров. Обозначение %0 в ротации не участвует и никак не изменяется.

Директива позволяет производить циклический сдвиг и в обратном направлении (влево), для этого следует задать её параметр отрицательным. Так, после отработки директивы

```
%rotate -1
```

%1 будет обозначать параметр, ранее стоявший самым последним, %2 станет обозначать параметр, ранее бывший первым (то есть имевший обозначение %1) и т. д.

Вспомним, что ранее (см. стр. 105) мы обещали написать макрос pcall, позволяющий в одну строчку сформировать вызов подпрограммы с любым количеством аргументов. Сейчас, имея в своём распоряжении макросы с переменным числом аргументов и директиву %rotate, мы готовы это сделать. Наш макрос, который мы назовём просто pcall, будет принимать на вход адрес процедуры (аргумент для команды call) и произвольное количество параметров, предназначенное для размещения в стеке. Мы будем, как и раньше, предполагать для простоты, что каждый параметр занимает ровно 4 байта. Напомним, что параметры должны быть помещены в стек в обратном порядке, начиная с последнего. Мы

добьёмся этого с помощью макроцикла `%rep` и директивы `%rotate -1`, которая на каждом шаге будет делать последний (на текущий момент) параметр параметром номер 1. Количество итераций цикла на единицу меньше, чем количество параметров, переданных в макрос, потому что первый из параметров является именем процедуры и его в стек заносить не надо. После этого цикла нам останется снова превратить последний параметр в первый (на этот раз это как раз и окажется самый первый из всех параметров, то есть адрес процедуры) и сделать `call`, а затем вставить команду `add` для очистки стека от параметров. Итак, пишем:

```
%macro pcall 1-*  ; от одного до скольки угодно
  %rep %0 - 1     ; цикл по всем параметрам кроме первого
    %rotate -1    ; последний параметр становится %1
        push dword %1
  %endrep
  %rotate -1      ; адрес процедуры становится %1
        call %1
        add esp, (%0 - 1) * 4
%endmacro
```

Если теперь вызвать этот макрос, например, вот так:

```
        pcall myproc, eax, myvar, 27
```

то результатом подстановки станет следующий фрагмент:

```
        push dword 27
        push dword myvar
        push dword eax
        call myproc
        add esp, 12
```

что, собственно, нам и требовалось.

§ 3.5.8. Макродирективы для работы со строками

Ассемблер NASM поддерживает две директивы, предназначенные для преобразования строк (строковых констант) во время макропроцессирования. Они могут оказаться полезными, например, внутри многострочного макроса, одним из параметров которого является (должна быть) строка и с этой строкой необходимо предварительно выполнить те или иные преобразования.

Первая из директив, `%strlen`, позволяет определить длину строки. Директива имеет два параметра. Первый из них — имя макропеременной, которой следует присвоить число, соответствующее длине строки, ну а второй — собственно строка. Так, в результате выполнения

```
%strlen sl 'my string'
```

макропеременная s1 получит значение 9.

Вторая директива, %substr, позволяет выделить из строки символ с заданным номером. Например, после выполнения

```
%substr var1 'abcd' 1
%substr var2 'abcd' 2
%substr var3 'abcd' 3
```

макропеременные var1, var2 и var3 получат значения 'a', 'b' и 'c' соответственно, то есть произойдёт то же самое, как если бы мы написали

```
%define var1 'a'
%define var2 'b'
%define var3 'c'
```

Всё это имеет смысл, как правило, только в случае, если в качестве аргумента директивы получают либо имя макропеременной, либо обозначение позиционного параметра в многострочном макросе.

Напомним, что все макродирективы отрабатывают во время макропроцессирования (**перед** компиляцией, то есть задолго до выполнения нашей программы), так что, разумеется, на момент соответствующих макроподстановок все используемые строки должны быть уже известны.

§ 3.6. Командная строка NASM

Рассказ об ассемблере NASM мы завершаем кратким обзором аргументов его командной строки. Как уже говорилось, при вызове программы nasm необходимо указать имя файла, содержащего исходный текст на языке ассемблера, а кроме этого, обычно требуется указать *ключи*, задающие режим работы. С некоторыми из них мы уже знакомы: это ключи -f, -o и -d.

Напомним, что ключ -f позволяет указать *формат* получаемого кода. В нашем случае всегда используется формат elf. Интересно, что, если не указать этот ключ, ассемблер создаст выходной файл в «сыром» формате, то есть, попросту говоря, переведёт наши команды в двоичное представление и в таком виде запишет в файл. Работая под управлением операционных систем, мы такой файл запустить на выполнение не сможем, однако если бы мы, к примеру, хотели написать программу для размещения в загрузочном секторе диска, то «сырой» формат оказался бы как раз тем, что нам нужно.

Ключ -o задаёт имя файла, в который следует записать результат трансляции. Если мы используем формат elf, то вполне можем доверить выбор имени файла самому NASM'у: он отбросит от имени исходного файла суффикс .asm и заменит его на .o, что нам в большинстве случаев и требуется. Если же по каким-то причинам нам удобнее другое имя, мы можем указать его явно с помощью -o.

Ключ -d, как мы уже знаем (см. стр. 112), используется для определения макросимвола в случае, если мы не хотим делать этого путём редактирования исходного текста. Мы использовали его в форме -dSYMBOL, что даёт тот же эффект, как если в начало программы вставить строку %define SYMBOL. Но можно использовать его и для задания *значения* макросимвола: например, -dSIZE=1024 не только определит символ SIZE, но и припишет ему значение 1024, как это сделала бы директива %define SIZE 1024.

Очень интересны в познавательном плане возможности генерации так называемого ***листинга*** — подробного отчёта ассемблера о проделанной работе. Листинг включает в себя строки исходного кода, снабжённые информацией об используемых адресах и о том, какой итоговый код сгенерирован в результате обработки каждой исходной строки. Генерация листинга запускается ключом -l, после которого требуется указать имя файла. Для примера возьмите любую программу на языке ассемблера и оттранслируйте её с флагом -l; так, если ваша программа называется prog.asm, попробуйте применить команду

```
nasm -f elf -l prog.lst prog.asm
```

в результате которой текст листинга будет помещён в файл prog.lst; обязательно просмотрите получившийся файл и задайте вашему преподавателю вопросы по поводу всего, что в листинге оказалось непонятно.

Весьма полезным может оказаться ключ -g, указывающий NASM'у на необходимость включения в результаты трансляции так называемой *отладочной информации*. При указании этого ключа NASM вставляет в объектный файл помимо объектного кода ещё и сведения об имени исходного файла, номерах строк в нём и т. п. Для работы программы вся эта информация совершенно бесполезна, тем более что по объёму она может в несколько раз превышать «полезный» объектный код. Однако в случае, если ваша программа работает не так, как вы от неё ожидаете, компиляция с флажком -g позволит вам воспользоваться отладчиком (например, gdb) для пошагового выполнения программы, что, в свою очередь, даст возможность разобраться в происходящем.

Ещё один полезный ключ — -e; он предписывает NASM'у прогнать наш исходный код через макропроцессор, выдать результат в поток стандартного вывода (попросту говоря, на экран) и на этом успокоиться. Такой режим работы может оказаться полезен, если мы ошиблись при написании макроса и никак не можем понять, в чём наша ошибка заключается; увидев результат макропроцессирования нашей программы, мы, скорее всего, сможем понять, что и почему пошло не так.

NASM поддерживает и другие ключи командной строки; желающие могут изучить их самостоятельно, обратившись к документации.

Глава 4. Взаимодействие с операционной системой

В этой главе мы рассмотрим средства взаимодействия пользовательской программы с операционной системой, что позволит в дальнейшем отказаться от использования макросов из файла `stud_io.inc`, а при желании и самостоятельно создавать их аналоги.

Пользовательские задачи обращаются к ядру операционной системы, используя так называемые ***системные вызовы***, которые, в свою очередь, реализованы через механизм ***программных прерываний***. Чтобы понять, что это такое, нам придётся подробно обсудить, что такое прерывания, какие они бывают и для чего служат, поэтому первые два параграфа этой главы мы посвятим изложению необходимых теоретических сведений, и лишь затем, имея готовую базу, рассмотрим механизм системных вызовов операционных систем Linux и FreeBSD на уровне машинных команд.

§ 4.1. Мультизадачность и её основные виды

§ 4.1.1. Понятие одновременности выполнения

Как уже говорилось во введении, ***мультизадачность*** или режим мультипрограммирования — это такой режим работы вычислительной системы, при котором несколько программ могут выполняться в системе одновременно. Для этого, вообще говоря, не нужно несколько физических *процессоров*. Вычислительная система может иметь всего один процессор, что не мешает само по себе реализации режима мультипрограммирования. Так или иначе, количество процессоров в системе в общем случае меньше количества одновременно выполняемых программ. Ясно, что процессор в каждый момент времени может выполнять только одну программу. Что же, в таком случае, понимается под мультипрограммированием? Кажущийся парадокс разрешается введением следующего

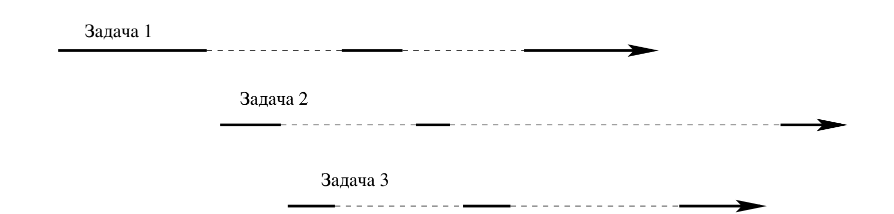


Рис. 4.1. Одновременное выполнение задач на одном процессоре

определения ***одновременности*** для случая выполняющихся программ (***процессов***, или ***задач***):

Две задачи, запущенные на одной вычислительной системе, называются выполняемыми *одновременно*, если периоды их выполнения (временно́й отрезок с момента запуска до момента завершения каждой из задач) полностью или частично перекрываются. Иными словами, если процессор, работая в каждый момент времени с одной задачей, при этом переключается между несколькими задачами, уделяя внимание то одной из них, то другой, эти задачи в соответствии с нашим определением будут считаться выполняемыми одновременно (см. рис. 4.1).

§ 4.1.2. Пакетный режим

В простейшем случае мультизадачность позволяет решить проблему простоя центрального процессора во время операций ввода-вывода. Представим себе вычислительную систему, в которой выполняется одна задача (например, обсчет сложной математической модели). В некоторый момент времени задаче может потребоваться операция обмена данными с каким-либо внешним устройством (например, чтение очередного блока входных данных либо, наоборот, запись конечных или промежуточных результатов).

Скорость работы внешних устройств (дисков и т. п.) обычно на порядки ниже, чем скорость работы центрального процессора, и в любом случае никоим образом не бесконечна. Так, для чтения заданного блока данных с диска необходимо включить привод головки, чтобы переместить её в нужное положение (на нужную дорожку) и дождаться, пока сам диск повернётся на нужный угол (для работы с заданным сектором); затем, пока сектор проходит под головкой, прочитать записанные в этом секторе данные во внутренний буфер контроллера диска[1]; наконец, следует разместить прочитанные данные в той области памяти, где

[1]Чтение непосредственно в оперативную память теоретически возможно, но технически сопряжено с определенными трудностями и применяется редко.

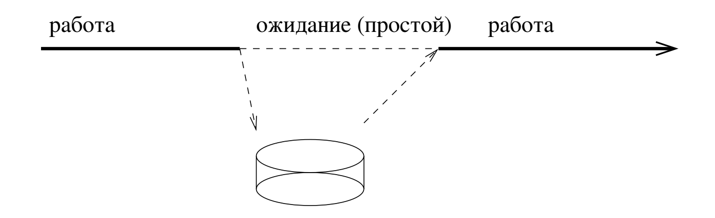


Рис. 4.2. Простой процессора в однозадачной системе

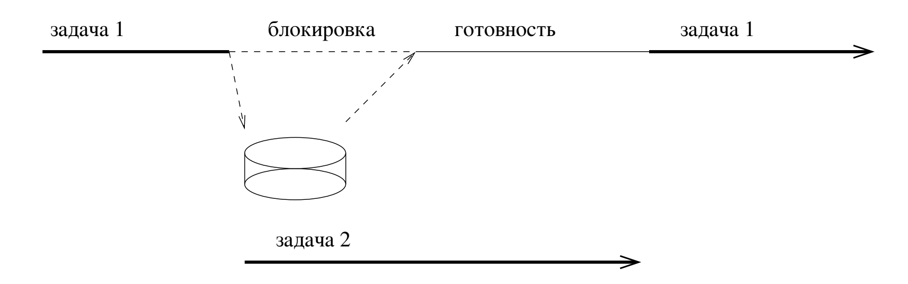


Рис. 4.3. Пакетная ОС

их появления ожидает пользовательская программа, и лишь после этого вернуть ей управление. Всё это время (как минимум, время, затрачиваемое на перемещение головки и ожидание нужной фазы поворота диска) центральный процессор будет простаивать (рис. 4.2). Если задача у нас всего одна и больше делать нечего, такой простой не создаёт проблем, но если кроме той задачи, которая уже работает, у нас есть и другие задачи, дожидающиеся своего часа, то лучше бы было употребить время центрального процессора, впустую пропадающее в ожидании окончания операций ввода-вывода, на решение других задач. Именно так поступают мультизадачные операционные системы. В такой системе из задач, которые нужно решать, формируется *очередь заданий*. Как только активная задача затребует проведение операции ввода-вывода, операционная система выполняет необходимые действия по запуску контроллеров устройств на исполнение запрошенной операции либо ставит запрошенную операцию в очередь, если начать её немедленно по каким-то причинам нельзя, после чего активная задача заменяется на другую — новую (взятую из очереди) или уже выполнявшуюся раньше, но не успевшую завершиться. Замененная задача в этом случае считается перешедшей в состояние ожидания результата ввода-вывода, или ***состояние блокировки***.

В простейшем случае новая активная задача остается в режиме выполнения до тех пор, пока она не завершится либо не затребует, в свою очередь, проведение операции ввода-вывода. При этом блокированная задача по окончании операции ввода-вывода переходит из состояния блокировки в ***состояние готовности к выполнению***, но переключения на нее не происходит (см. рис. 4.3); это обусловлено тем, что операция смены активной задачи, вообще говоря, отнимает много процессорного времени. Такой способ построения мультизадачности, при котором смена активной задачи происходит только в случае ее окончания или запроса на операцию ввода-вывода, называется ***пакетным режимом***[2], а операционные системы, реализующие этот режим,— ***пакетными операционными системами***. Режим пакетной мультизадачности является самым эффективным с точки зрения использования вычислительной мощности центрального процессора, поэтому именно пакетный режим используется для управления суперкомпьютерами и другими машинами, основное назначение которых — большие объемы численных расчетов.

§ 4.1.3. Режим разделения времени

С появлением первых терминалов и диалогового (иначе говоря, интерактивного) режима работы с компьютерами возникла потребность в других стратегиях смены активных задач, или, как принято говорить, ***планирования времени центрального процессора***. Действительно, пользователю, ведущему диалог с той или иной программой, вряд ли захочется ждать, пока некая активная задача, вычисляющая, скажем, обратную матрицу порядка 1000x1000, завершит свою работу. При этом много процессорного времени на обслуживание диалога с пользователем не требуется: в ответ на каждое действие пользователя (например, нажатие на клавишу) обычно необходимо выполнить набор действий, укладывающийся в несколько миллисекунд, тогда как самих таких событий пользователь даже в режиме активного набора текста может создать никак не больше трех-четырех в секунду (скорость компьютерного набора 200 символов в минуту считается очень высокой). Соответственно, было бы нелогично ждать, пока пользователь полностью завершит свой диалоговый сеанс: бóльшую часть времени процессор мог бы производить арифметические действия, необходимые для задачи, вычисляющей матрицу. Решить проблему позволяет ***режим разделения времени***. В этом режиме каждой задаче отводится определенное время работы,

[2]Русскоязычный термин «пакетный режим» является устоявшимся, хотя и не слишком удачным переводом английского термина «batch mode»; слово *batch* можно также перевести как «колода» (собственно, изначально имелись в виду колоды перфокарт, олицетворявшие задания). Не следует путать этот термин со словами, происходящими от английского слова packet, которое тоже обычно переводится на русский как «пакет».

называемое ***квантом времени***. По окончании этого кванта, если в системе имеются другие готовые к исполнению задачи, активная задача принудительно приостанавливается и заменяется другой задачей. Приостановленная задача помещается в ***очередь задач, готовых к выполнению*** и находится там, пока остальные задачи отработают свои кванты; затем она снова получает очередной квант времени для работы, и т. д. Естественно, если активная задача затребовала операцию ввода-вывода, она переводится в состояние блокировки (точно так же, как и в пакетном режиме). Задачи, находящиеся в состоянии блокировки, не ставятся в очередь на выполнение и не получают квантов времени до тех пор, пока операция ввода-вывода не будет завершена (либо не исчезнет другая причина блокировки), и задача не перейдет в состояние готовности к выполнению.

Существуют различные алгоритмы поддержки очереди на выполнение, в том числе и такие, в которых задачам приписывается некоторый приоритет, выраженный числом. Например, в ОС Unix обычно задача имеет две составляющие приоритета — статическую и динамическую; статическая составляющая представляет собой заданный администратором уровень «важности» выполнения данной конкретной задачи, динамическая же изменяется планировщиком: пока задача находится в стадии выполнения, её динамический приоритет падает, когда же задача находится в очереди на исполнение, динамическая составляющая приоритета, напротив, растёт. Из нескольких готовых к исполнению задач выбирается имеющая наибольшую сумму приоритетов, так что рано или поздно задача даже с самым низким статическим приоритетом получит управление за счет возросшего динамического приоритета.

Некоторые операционные системы, включая ранние версии Windows, применяли стратегию, занимающую промежуточное положение между пакетным режимом и режимом разделения времени. В этих системах задачам выделялся квант времени, как и в системах разделения времени, но принудительной смены текущей задачи по истечении кванта времени не производилось; система проверяла, не истек ли квант времени у текущей задачи, только когда задача обращалась к операционной системе за какими-либо услугами (не обязательно за вводом-выводом). Таким образом, задача, не нуждающаяся в услугах операционной системы, могла оставаться на процессоре сколь угодно долго, как и в пакетных операционных системах. Такой режим работы называется ***невытесняющим***. В современных системах он не применяется, поскольку налагает слишком жесткие требования на исполняемые в системе программы; так, в ранних версиях Windows любая программа, занятая длительными вычислениями, блокировала работу всей системы.

§ 4.1.4. Режим реального времени

Иногда режим разделения времени также оказывается непригоден. В некоторых ситуациях, таких как управление полетом самолета, ядерным реактором, ав-

томатической линией производства и т. п., некоторые задачи должны быть завершены строго до определенного момента времени; так, если автопилот самолета, получив сигнал от датчиков тангажа и крена, потратит на вычисление необходимого корректирующего воздействия больше времени, чем допустимо, самолет может вовсе потерять управление.

В случае, когда выполняемые задачи (как минимум некоторые из них) имеют жесткие рамки по необходимому времени завершения, применяются ***операционные системы реального времени***. В отличие от систем разделения времени, задача планировщика реального времени не в том, чтобы дать всем программам отработать некоторое время, а в том, чтобы *обеспечить завершение каждой задачи за отведённое ей время*, если же это невозможно — снять задачу, освободив процессор для тех задач, которые ещё можно успеть завершить к сроку.

§ 4.1.5. Аппаратная поддержка мультизадачности

Ясно, что для построения мультизадачного режима работы вычислительной системы аппаратура (прежде всего сам центральный процессор) должна обладать определенными свойствами. О некоторых из них мы уже говорили в § 1.2 — это, во-первых, защита памяти, а во-вторых, разделение машинных команд на обычные и привилегированные, с отключением возможности выполнения привилегированных команд в ограниченном режиме работы центрального процессора.

Действительно, при одновременном нахождении в памяти машины нескольких программ, если не предпринять специальных мер, одна из программ может модифицировать данные или код других программ или самой операционной системы. Даже если допустить отсутствие злого умысла у разработчиков всех запускаемых программ, от случайных ошибок в программах нас это допущение не спасет, причём такая ошибка может, с одной стороны, привести к тяжелым авариям всей системы, а с другой стороны — оказаться совершенно неуловимой, вплоть до абсолютной невозможности установить, какая из задач «виновата» в происходящем. Дело в том, что для обнаружения и устранения ошибки необходима возможность воссоздания обстоятельств, при которых эта ошибка проявляется, а точно воссоздать состояние всей системы со всеми запущенными в ней задачами практически невозможно. Очевидно, необходимы средства ограничения возможностей работающей программы по доступу к областям памяти, занятым другими программами. Программно такую защиту можно реализовать разве что путем интерпретации всего машинного кода исполняющейся программы, что, как правило, недопустимо из соображений эффективности. Таким образом, необходима *аппаратная* поддержка защиты памяти, позволяющая ограничить возможности текущей задачи по доступу к оперативной памяти.

Коль скоро существует защита памяти, процессор должен иметь набор команд для управления этой защитой. Если, опять-таки, не пред-

принять специальных мер, то такие команды сможет исполнить любая из выполняющихся программ, сняв защиту памяти или модифицировав ее конфигурацию, что сделало бы саму защиту памяти практически бессмысленной. Рассматриваемая проблема касается не только защиты памяти, но и работы с внешними устройствами. Как уже говорилось, чтобы обеспечить нормальное взаимодействие всех программ с устройствами ввода-вывода, операционная система должна взять непосредственную работу с устройствами на себя, а пользовательским программам предоставлять интерфейс для обращения к операционной системе за услугами по работе с устройствами, причём пользовательские программы должны иметь возможность работы с внешними устройствами только через операционную систему. Соответственно, необходимо запретить пользовательским программам выполнение команд процессора, осуществляющих чтение/запись портов ввода-вывода. Вообще, передавая управление пользовательской программе, операционная система должна быть уверена, что задача не сможет (иначе как путем обращения к самой операционной системе) выполнить никакие действия, влияющие на систему в целом.

Проблема решается введением двух режимов работы центрального процессора: ***привилегированного*** и ***ограниченного***. В литературе привилегированный режим часто называют «режимом ядра» или «режимом супервизора» (англ. ***kernel mode***, ***supervisor mode***). Ограниченный режим называют также «пользовательским режимом» (англ. ***user mode***) или просто ***непривилегированным*** (англ. ***nonprivileged***). Термин ***ограниченный режим*** избран в нами как наиболее точно описывающий сущность этого режима работы центрального процессора без привязки к его использованию операционными системами. В привилегированном режиме процессор может выполнять любые существующие команды. В ограниченном режиме выполнение команд, влияющих на систему в целом, запрещено; разрешаются только команды, эффект которых ограничен модификацией данных в областях памяти, не закрытых защитой памяти. Сама операционная система выполняется в привилегированном режиме, пользовательские программы — в ограниченном.

Как мы уже отмечали в §1.2, пользовательская программа может только модифицировать данные в отведённой ей памяти; любые другие действия требуют обращения к операционной системе. Это обеспечивается поддержкой в центральном процессоре механизма защиты памяти и наличием ограниченного режима работы. Соблюдения этих двух аппаратных требований, однако, ещё не достаточно для реализации мультизадачного режима работы системы.

Вернемся к ситуации с операцией ввода-вывода. В однозадачной системе (рис. 4.2 на стр. 125) во время исполнения операции ввода-вывода центральный процессор мог непрерывно опрашивать контроллер устрой-

ства на предмет его готовности (завершена ли требуемая операция), после чего произвести необходимые действия по подготовке к возобновлению работы активной задачи — в частности, скопировать прочитанные данные из буфера контроллера в область памяти, в которой задача ожидает появления данных. Следует отметить, что в этом случае процессор был бы непрерывно занят во время операции ввода-вывода, несмотря на то, что никаких полезных вычислений он при этом не производил. Такой способ взаимодействия называется ***активным ожиданием***. Ясно, что активное ожидание неэффективно, так как процессорное время можно было бы использовать с бо́льшей пользой. При переходе к мультизадачной обработке, показанной на рис. 4.3 на стр. 125, возникает определенная проблема. В момент завершения операции ввода-вывода процессор занят исполнением второй задачи. Между тем, в момент завершения операции требуется как минимум перевести первую задачу из состояния блокировки в состояние готовности; более того, могут потребоваться и другие действия, такие как копирование данных из буфера контроллера, сброс контроллера (например, выключение мотора диска), а в более сложных ситуациях — инициирование другой операции ввода-вывода, ранее отложенной (это может быть операция чтения с того же диска, которую затребовала другая задача в то время, как первая операция еще выполнялась). Проблема состоит в том, каким образом операционная система узнает о завершении операции ввода-вывода, если процессор при этом занят выполнением другой задачи и непрерывного опроса контроллера не производит.

Решить проблему позволяет аппарат ***прерываний***. В данном конкретном случае в момент завершения операции контроллер подает центральному процессору определенный сигнал (электрический импульс), называемый ***запросом прерывания***. Центральный процессор, получив этот сигнал, прерывает выполнение активной задачи и передает управление процедуре операционной системы, которая выполняет все действия, необходимые по окончании операции ввода-вывода. Такая процедура называется ***обработчиком прерывания***. После завершения процедуры-обработчика управление возвращается активной задаче.

Для реализации пакетного мультизадачного режима достаточно, чтобы на уровне аппаратуры были реализованы прерывания, защита памяти и два режима работы процессора. Если же необходимо реализовать систему разделения времени или реального времени, требуется наличие в аппаратуре еще одного компонента — ***таймера***. Действительно, планировщику операционной системы разделения времени нужна возможность отслеживания истечения квантов времени, выделенных пользовательским программам; в системе реального времени такая возможность также необходима, причем требования к ней даже более жёсткие: не сняв вовремя с процессора активное на тот момент приложение, планировщик

рискует не успеть выделить более важным программам необходимое им процессорное время, в результате чего могут наступить неприятные последствия (вспомните пример с автопилотом самолёта). Таймер представляет собой сравнительно простое устройство, вся функциональность которого сводится к генерации прерываний через равные промежутки времени. Эти прерывания дают возможность операционной системе получить управление, проанализировать текущее состояние имеющихся задач и при необходимости сменить активную задачу.

Итак, для реализации мультизадачной операционной системы аппаратное обеспечение компьютера обязано поддерживать:

- аппарат прерываний;
- защиту памяти;
- привилегированный и ограниченный режимы работы центрального процессора;
- таймер.

Первые три свойства необходимы в любой мультизадачной системе, последнее может отсутствовать в случае пакетной планировки (хотя в реально существующих системах таймер присутствует всегда). Следует обратить внимание, что из перечисленных свойств только таймер является отдельным устройством, остальные три представляют собой особенности центрального процессора.

Теоретически при наличии таймера можно сделать прерывание по таймеру *единственным* прерыванием в системе. В этом случае операционная система, получив управление в результате такого прерывания, должна будет уже сама опросить все активные контроллеры внешних устройств на предмет завершения выполнявшихся операций, а также проверить, не находится ли активная задача в каком-то «специальном» состоянии, обозначающем потребность в системном вызове. Реально такая схема порождает множество проблем, прежде всего с эффективностью, а выигрыш от её применения неочевиден.

§ 4.2. Виды прерываний

Современный термин «***прерывание***» довольно далеко ушел в своем развитии от изначального значения; начинающие программисты часто с удивлением обнаруживают, что некоторые прерывания вовсе ничего не прерывают. Дать строгое определение прерывания было бы несколько затруднительно. Вместо этого попытаемся объяснить сущность различных видов прерываний и найти между ними то общее, что и оправдывает существование самого термина.

§ 4.2.1. Внешние (аппаратные) прерывания

Прерывания в изначальном смысле уже знакомы нам из предыдущего параграфа. Те или иные устройства вычислительной системы могут осуществлять свои функции независимо от центрального процессора; в этом случае им может время от времени требоваться внимание операционной системы, но единственный центральный процессор (или, что ничуть не лучше, все имеющиеся в системе центральные процессоры) может быть именно в такой момент занят обработкой пользовательской программы. Аппаратные (или ***внешние***) прерывания были призваны решить эту проблему. Для поддержки аппаратных прерываний процессор имеет специально предназначенные для этого контакты; электрический импульс, поданный на такой контакт, воспринимается процессором как сигнал о том, что некоторому устройству требуется внимание операционной системы. В современных архитектурах, основанных на общей шине, для запроса прерывания используется одна из дорожек шины.

Последовательность событий при возникновении и обработке прерывания выглядит приблизительно следующим образом[3]:

1. Устройство, которому требуется внимание процессора, устанавливает на шине сигнал «запрос прерывания».

2. Процессор доводит выполнение текущей программы до такой точки, в которой выполнение можно прервать так, чтобы потом восстановить его с того же места; после этого процессор выставляет на шине сигнал «подтверждение прерывания». При этом другие прерывания блокируются.

3. Получив подтверждение прерывания, устройство передает по шине некоторое число, идентифицирующее данное устройство; это число называют ***номером прерывания***.

4. Процессор сохраняет где-то (обычно в стеке активной задачи) текущие значения счетчика команд и регистра флагов; это называется ***малым упрятыванием***. Счетчик команд и регистр флагов должны быть сохранены по той причине, что выполнение первой же инструкции обработчика прерывания изменит (испортит) и то, и другое, сделав невозможным прозрачный (т. е. незаметный для пользовательской задачи) возврат из обработчика; остальные регистры обработчик прерывания может при необходимости сохранить самостоятельно.

5. Устанавливается привилегированный режим работы центрального процессора, после чего управление передается на точку входа про-

[3]Здесь приводится общая схема; в действительности все намного сложнее.

цедуры в операционной системе, называемой, как мы уже говорили, ***обработчиком прерывания***. Адрес обработчика может быть предварительно считан из специальных областей памяти, либо вычислен иным способом.

Напомним, что переключение из привилегированного режима работы центрального процессора в ограниченный можно осуществить простой командой, поскольку в привилегированном режиме доступны все возможности процессора; в то же время, переход из ограниченного (пользовательского) режима обратно в привилегированный произвести с помощью обычной команды нельзя, поскольку это лишило бы смысла само существование привилегированного и ограниченного режимов. В этом плане **прерывание интересно ещё и тем, что при его возникновении режим работы центрального процессора становится привилегированным**.

§ 4.2.2. Внутренние прерывания (ловушки)

Чтобы понять, о чем пойдет речь в этом параграфе, рассмотрим следующий вопрос: что следует делать центральному процессору, если активная задача выполнила целочисленное деление на ноль? Ясно, что дальнейшее выполнение программы лишено смысла: результат деления на ноль невозможно представить каким-либо целым числом, так что в переменной, которая должна была содержать результат произведённого деления, в лучшем случае будет содержаться мусор; соответственно, и конечные результаты, скорее всего, окажутся иррелевантными. Пытаться оповестить программу о происшедшем путем выставления какого-нибудь флага, очевидно, также бессмысленно. Если программист не произвел **перед** выполнением деления проверку делителя на равенство нулю, представляется и вовсе ничтожной вероятность того, что он станет проверять **после** деления значение какого-то флага.

Завершить текущую задачу процессор самостоятельно не может. Это слишком сложное действие, зависящее от реализации операционной системы. Остается только один вариант: передать управление операционной системе, известив её о происшедшем. Что делать с аварийной задачей, операционная система решит самостоятельно. Для этого требуется, очевидно, переключиться в привилегированный режим и передать управление коду операционной системы; перед этим желательно сохранить регистры (хотя бы счётчик команд и регистр флагов); даже если задача ни при каких условиях не будет продолжена с того же места (а предполагать это процессор, вообще говоря, не вправе), значения регистров в любом случае могут пригодиться операционной системе для анализа происшествия. Более того, каким-то образом следует сообщить операционной системе о причине того, что управление передано ей; кроме деления

на ноль, такими причинами могут быть нарушение защиты памяти, попытка выполнить запрещённую или несуществующую инструкцию и т. п.

Легко заметить, что действия, которые должен выполнить процессор, оказываются очень похожи на рассмотренный ранее случай аппаратного прерывания. Основное отличие состоит в отсутствии обмена по шине (запроса и подтверждения прерывания): действительно, информация о перечисленных событиях возникает внутри процессора, а не вне его[4]. Остальные шаги по обработке деления на ноль и других подобных ситуаций повторяют шаги по обработке аппаратного прерывания практически дословно. Поэтому обработку ситуаций, в которых дальнейшее выполнение активной задачи оказывается невозможной по причине выполненных ею некорректных действий, называют так же, как и действия по запросу внешних устройств — прерываниями. Чтобы не путать разные по своей природе прерывания, их делят на внешние (аппаратные) и внутренние; такая терминология оправдана тем, что причина внешнего прерывания находится вне центрального процессора, тогда как причина внутреннего — у ЦП внутри. Иногда внутренние прерывания называют иначе, например ***ловушками*** (traps), исключениями (exceptions) или как-то ещё.

§ 4.2.3. Программные прерывания

Как уже говорилось, пользовательской задаче не позволяется делать ничего, кроме преобразования данных в отведённой ей памяти. Все действия, затрагивающие внешний по отношению к задаче мир, выполняются через операционную систему. Соответственно, необходим механизм, позволяющий пользовательской задаче обратиться к ядру операционной системы за теми или иными услугами. Напомним, что **обращение пользовательской задачи к ядру операционной системы за услугами называется *системным вызовом***. Ясно, что по своей сути системный вызов — это передача управления от пользовательской задачи ядру операционной системы. Однако здесь есть две проблемы. Во-первых, ядро работает в привилегированном режиме, а пользовательская задача — в ограниченном. Во-вторых, пространство адресов ядра для пользовательской задачи обычно недоступно (более того, в адресном пространстве задачи этих адресов может вообще не быть). Впрочем, даже если бы оно было доступно, позволить пользовательской задаче передавать управление в произвольную точку ядра было бы несколько странно.

Итак, для осуществления системного вызова необходимо сменить режим выполнения с пользовательского на привилегированный и передать

[4] С точки зрения реализации внутренние прерывания могут оказаться многократно проще, чем аппаратные, за счет того, что они всегда происходят на определенной фазе выполнения инструкции; подробности читатель найдет в книге [1].

управление в некоторую точку входа в операционной системе. Нам уже известны два случая, в которых происходит что-то подобное — это аппаратные и внутренние прерывания. Изобретать дополнительный механизм для системного вызова не обязательно: для его реализации можно использовать частный случай внутреннего прерывания, инициируемый специально предназначенной для этого машинной инструкцией. На разных архитектурах соответствующая инструкция может называется `trap` (ловушка), `svc` (supervisor call, то есть «обращение к супервизору») и т. д. Рассматриваемые нами процессоры семейства i386 используют команду `int` (от слова interrupt — прерывание). Такое прерывание называется ***программным прерыванием***. Отличие этого вида прерывания от остальных состоит в том, что оно происходит по инициативе пользовательской задачи, тогда как другие прерывания случаются без её ведома: внешние — по требованию внешних устройств, внутренние — в случае непредвиденных обстоятельств, которые вряд ли были выполняемой программой предусмотрены. Некоторые авторы не делают различия между терминами «программное прерывание» и «системный вызов», называя системным вызовом как само обращение к ОС, так и программное прерывание, используемое для его осуществления.

Некоторые процессоры могут предусматривать и иные механизмы передачи управления операционной системе. Так, процессоры семейства i386 реализуют так называемые *шлюзы* (англ. gates) для передачи управления привилегированным программам с одновременным повышением уровня привилегированности режима работы процессора, а самих этих уровней, называемых *кольцами защиты*, процессоры семейства i386 поддерживают не два, а четыре; впрочем, операционные системы этим обычно не пользуются.

Так или иначе, повышение уровня привилегий (переход из ограниченного режима в привилегированный) возможно только при условии одновременной передачи управления на заранее заданную точку входа, причем адреса возможных точек входа могут настраиваться только в привилегированном режиме. Таким образом, операционная система имеет возможность гарантировать, что при смене режима работы на привилегированный управление получит только код самой операционной системы, причем только такой её код, который для этого специально предназначен. Исполнение в привилегированном режиме какого бы то ни было пользовательского кода полностью исключается.

§ 4.3. Системные вызовы в ОС Unix

Перейдём теперь к освоению системных вызовов на практике. Следует отметить, что соглашения о том, как конкретно должен происхо-

дить системный вызов, как передать ему необходимые параметры, какое использовать прерывание, как получить результат выполнения и т. п., варьируются от системы к системе. Даже если речь идёт о двух представителях семейства Unix (ОС Linux и ОС FreeBSD), работающих на одной и той же аппаратной платформе i386, низкоуровневая реализация системных вызовов оказывается в них совершенно различна. Следующие два параграфа будут посвящены описанию соглашений об организации системных вызовов этих двух систем[5]; при желании вы можете прочитать только один из этих двух параграфов, относящийся к той системе, которую вы используете.

Следует иметь в виду, что системы семейства Unix расчитаны в основном на программирование на языке Си. Естественно, для этого языка вместе с системой поставляются библиотеки, облегчающие работу с системными вызовами — в частности, для каждого системного вызова предоставляется библиотечная функция, позволяющая обратиться к услугам ядра как к обычной подпрограмме. Системные вызовы в ОС Unix имеют названия, совпадающие с именами соответствующих функций-обёрток из библиотеки языка Си. К сожалению, такая ориентированность на Си приводит к некоторым неудобствам при работе на уровне языка ассемблера. Так, системные вызовы при переходе от системы к системе могут менять свои номера (например, `getppid` в ОС Linux имеет номер 64, а в ОС FreeBSD — номер 39). Программисты, работающие на языке Си, об этом могут не задумываться, поскольку в любой системе семейства Unix им достаточно вызвать обычную функцию с именем `getppid`, а конкретное исполнение системного вызова возлагается на библиотеку, которая прилагается к системе, так что программа, написанная программистом на Си с использованием `getppid`, будет успешно компилироваться на любой системе и работать одинаково. Иное дело, если мы пишем на языке ассемблера. Никакой библиотеки системных вызовов у нас при этом нет, номер вызова мы должны указать в программе явно, так что в тексте, предназначенном для Linux, придётся использовать число `64`, тогда как для FreeBSD нужно будет число `39`. Получается, что написанный нами исходный текст будет пригоден для одной системы и ошибочен для другой. Аналогично обстоят дела и с некоторыми числовыми константами, которые вызовы получают на вход.

Частично нас может выручить макропроцессор с его директивами условной компиляции, либо мы можем ограничиться только одной системой (что, на самом деле, не совсем правильно). К счастью, системы FreeBSD и Linux всё же во многом похожи друг на друга и числовые значения, связанные с системными вызовами, частично совпадают (с дру-

[5]Естественно, ОС Linux рассматривается в варианте для i386; версии этой системы, предназначенные для других аппаратных архитектур, устроены иначе.

гими системами семейства Unix было бы хуже). Так или иначе, кто предупреждён, тот вооружён.

§ 4.3.1. Конвенция ОС Linux

Ядро Linux на платформе i386 использует для осуществления системного вызова прерывание с номером 80h. Номер системного вызова передаётся ядру через регистр EAX; если системный вызов принимает параметры, то они располагаются, соответственно, в регистрах EBX, ECX, EDX, ESI и EDI; отметим, что все параметры системных вызовов являются четырёхбайтными значениями — либо целочисленными, либо адресными. Результат выполнения вызова возвращается через регистр EAX, причём значение, заключённое между fffff000h и ffffffffh, свидетельствует о происшедшей ошибке (и представляет собой условный код этой ошибки).

Рассмотрим для примера системный вызов write, позволяющий произвести вывод данных через один из открытых потоков ввода-вывода, в том числе запись в открытый файл, а также печать на стандартный вывод (в просторечии «на экран»). Этот системный вызов имеет номер 4 и принимает три параметра: дескриптор (номер) потока ввода-вывода, адрес памяти, где расположены данные, подлежащие выводу, и количество этих данных в байтах. Отметим, что поток стандартного вывода в ОС Unix имеет дескриптор 1 (точнее, поток вывода под номером 1 считается стандартным выводом). Таким образом, если мы хотим вывести строку «на экран», то есть сделать то, что делает макрос PRINT, нам нужно будет занести число 4 в EAX, занести число 1 в EBX, занести адрес строки в ECX и длину строки — в EDX, а затем дать команду int 80h, чтобы инициировать программное прерывание.

Другой важный системный вызов — это вызов _exit, используемый для завершения программы. Он имеет номер 1 и принимает один параметр, представляющий собой ***код завершения***. Программы используют код завершения, чтобы сообщить операционной системе, успешно ли они справились с возложенной на них задачей: если всё прошло как ожидалось, используется код 0, если же в ходе работы возникли те или иные ошибки, используются коды 1, 2 и т. д.

Зная всё это, мы можем написать программу, печатающую строку и сразу после этого завершающуюся; файл stud_io.inc и его макросы нам для этого больше не нужны:

```
global  _start

section .data
msg     db "Hello world", 10
msg_len equ $-msg
```

```
section .text
_start: mov     eax, 4          ; вызов write
        mov     ebx, 1          ; стандартный вывод
        mov     ecx, msg
        mov     edx, msg_len
        int     80h

        mov     eax, 1          ; вызов _exit
        mov     ebx, 0          ; код "успех"
        int     80h
```

§ 4.3.2. Конвенция ОС FreeBSD

Описание конвенции ОС FreeBSD несколько сложнее. Эта система также использует прерывание `80h` и принимает номер системного вызова через регистр `EAX`, но все параметры вызова передаются не через регистры, а через стек, подобно тому, как передаются параметры в подпрограммы в соответствии с соглашениями языка Си, то есть в обратном порядке (см. стр. 82). Как и в ОС Linux, все параметры вызовов представляют собой четырёхбайтные значения. Результат выполнения системного вызова возвращается через регистр `EAX`, но при этом о происшедшей ошибке свидетельствует не попадание значения в специальный промежуток (как это сделано в Linux), а установленное значение флага `CF`. Если `CF` сброшен, то вызов завершился успешно и его результат находится в `EAX`, если же флаг установлен, то произошла ошибка и в `EAX` записан код этой ошибки.

Необходимо отметить ещё одну особенность. Ядро FreeBSD предполагает, что управление ему передано путём обращения к процедуре следующего вида:

```
kernel:
        int 80h
        ret
```

Если у нас есть такая процедура, нам для обращения к ядру достаточно поместить в стек параметры точно так же, как для обычной процедуры, занести номер вызова в `EAX` и сделать `call kernel`; при этом команда `call` занесёт в стек адрес возврата, который и будет лежать на вершине стека в момент выполнения программного прерывания, а параметры будут располагаться в стеке ниже вершины. Ядро FreeBSD учитывает это и ничего не делает с числом на вершине стека (ведь это число — адрес возврата из процедуры `kernel` — никакого отношения к параметрам вызова не имеет), а настоящие параметры извлекает из стека ниже вершины (из позиций `[esp+4]`, `[esp+8]` и т. д.)

При работе на языке ассемблера выделять вызов прерывания в отдельную подпрограмму не обязательно, достаточно перед командой int занести в стек дополнительное «двойное слово», например, выполнив лишний раз команду push eax (или любой другой 32-битный регистр). Естественно, после выполнения системного вызова и возврата из него необходимо убрать из стека всё, что туда было занесено; делается это, как и при вызове обычных подпрограмм, путём увеличения регистра ESP на нужную величину простой командой add.

Описывая в предыдущем параграфе конвенцию ОС Linux, мы для иллюстрации использовали вызовы write и _exit (см. стр. 137). Аналогичная программа для FreeBSD будет выглядеть следующим образом:

```
global  _start

section .data
msg     db "Hello world", 10
msg_len equ $-msg

section .text
_start:
        push    dword msg_len
        push    dword msg
        push    dword 1         ; стандартный вывод
        mov     eax, 4          ; write
        push    eax             ; что угодно
        int     80h
        add     esp, 16         ; 4 двойных слова

        push    dword 0         ; код "успех"
        mov     eax, 1          ; вызов _exit
        push    eax             ; что угодно
        int     80h
```

Мы не стали очищать стек после системного вызова _exit, поскольку он всё равно не возвращает управление.

В этом примере мы не обрабатываем ошибки, предполагая, что запись в стандартный поток ввода всегда успешна (это в общем случае не так, но достаточно часто программисты на это не обращают внимания). Если бы мы хотели обрабатывать ошибки «честно», первой же командой после int 80h должна была бы быть команда jc или jnc, делающая условный переход в зависимости от состояния флага CF, в противном случае мы рискуем, что очередная команда выставит этот флаг сообразно своим результатам и признак происшедшей ошибки будет потерян. В ОС Linux с этим было несколько проще, достаточно не трогать регистр EAX, и ничего не потеряется.

§ 4.3.3. Некоторые системные вызовы Unix

В вышеприведённых примерах мы рассмотрели системные вызовы `_exit` и `write`; напомним, что `_exit` имеет[6] номер 1 и принимает один параметр — код завершения, а вызов `write` имеет номер 4 и принимает три параметра, а именно номер дескриптора потока вывода (1 для потока стандартного вывода), адрес области памяти, где расположены выводимые данные, и количество этих данных.

Для ввода данных (как из файлов, так и из стандартного потока ввода, т. е. «с клавиатуры») используется вызов `read`, имеющий номер 3. Его параметры аналогичны вызову `write`: первый параметр — номер дескриптора потока ввода (для стандартного ввода используется дескриптор 0), второй параметр — адрес области памяти, в которой необходимо разместить прочитанные данные, а третий — количество байтов, которое надлежит попытаться прочитать. Естественно, область памяти, адрес которой мы передаём вторым параметром, должна иметь размер не менее числа, передаваемого третьим параметром. **Очень важно проанализировать значение, возвращаемое вызовом `read`!** (напомним, что это значение сразу после вызова содержится в регистре `EAX`.) Если чтение прошло успешно, вызов вернёт строго положительное число — количество прочитанных байтов, которое, естественно, не может превышать «заказанное» через третий параметр количество, но вполне может оказаться меньше (например, мы потребовали прочитать 200 байтов, а реально было прочитано только 15). Очень важен случай, когда `read` возвращает число 0 — это свидетельствует о том, что в используемом потоке ввода возникла ситуация «конец файла». При чтении из файлов это значит, что весь файл прочитан и больше в нём данных нет. Однако «конец файла» может произойти не только при чтении из настоящего файла; так, при вводе с клавиатуры в ОС Unix можно сымитировать ситуацию «конец файла», нажав комбинацию клавиш Ctrl-D.

Помните, что программа, в которой используется вызов `read` и не производится анализ его результата, заведомо не может быть правильной. Действительно, мы в этом случае не можем знать, сколько первых байтов нашей области памяти содержат реально прочитанные данные, а сколько оставшихся продолжают содержать произвольный «мусор» — а значит, какая-либо осмысленная работа с этими данными невозможна.

При чтении, как и при использовании других системных вызовов, может произойти ошибка. В ОС Linux это легко обнаружить по отрицательному значению регистра `EAX` после возврата из вызова; в ОС FreeBSD для указания на то, что произошла ошибка, системные вызовы исполь-

[6] Во всяком случае, в системах Linux и FreeBSD; в дальнейшем, если нет явных указаний, подразумевается, что сказанное верно как минимум для этих двух систем.

зуют флаг CF (carry flag): если вызов завершился успешно, на выходе из него этот флаг будет сброшен, если же произошла ошибка, то флаг будет установлен. Это касается и вызова read, и рассмотренного ранее вызова write (мы не обрабатывали ошибочные ситуации, чтобы не усложнять наши примеры, но это не значит, что ошибки не могут произойти), и всех остальных системных вызовов.

На момент запуска программы для неё, как правило, открыты потоки ввода-вывода с номерами 0 (стандартный ввод), 1 (стандартный вывод) и 2 (поток для выдачи сообщений об ошибках), так что мы можем применять вызов read к дескриптору 0, а к дескрипторам 1 и 2 — вызов write. Часто, однако, задача требует создания иных потоков ввода-вывода, например, для чтения и записи файлов на диске. Прежде чем мы сможем работать с файлом, его необходимо *открыть*, в результате чего у нас появится ещё один поток ввода-вывода со своим номером (дескриптором). Делается это с помощью системного вызова open, имеющего номер 5. Вызов принимает три параметра. Первый параметр — адрес строки текста, задающей имя файла; имя должно заканчиваться нулевым байтом, который служит в качестве ограничителя. Второй параметр — число, задающее режим использования файла (чтение, запись и пр.); значение этого параметра формируется как битовая строка, в которой каждый бит означает определённую особенность режима, например, доступность только на запись, разрешение создать новый файл, если его нет, и т. п. К сожалению, расположение этих битов различно для ОС Linux и ОС FreeBSD; некоторые из флагов вместе с их описаниями и численными значениями приведены в таблице 4.1. Отметим, что наиболее часто встречаются два варианта для этого параметра. Первый из них — открытие файла только для чтения, в обеих рассматриваемых системах этот случай задаётся числом 0. Второй случай — открытие файла на запись, при котором файл создаётся, если его не было, а если он был, то его старое содержимое теряется (в программах на Си это задаётся комбинацией O_WRONLY|O_CREAT|O_TRUNC). Для Linux соответствующее числовое значение — 241h, для FreeBSD — 601h. Третий параметр вызова open используется только в случае создания файла и задаёт *права доступа* для него. Подробное описание этого параметра мы опускаем, отметим только, что в большинстве случаев его следует задать равным восьмеричному числу 0666q.

Для вызова open особенно важен анализ его возвращаемого значения и проверка, не произошла ли ошибка. Вызов open может завершиться с ошибкой в силу массы причин, большинство из которых программист никак не может ни предотвратить, ни предсказать: например, кто-то может неожиданно стереть файл, который мы собирались открыть на чтение, или запретить нам доступ к директории, где мы намеревались создать новый файл. Итак, после выполнения вызова open нам необхо-

название	описание	значение для	
		Linux	FreeBSD
O_RDONLY	только чтение	000h	000h
O_WRONLY	только запись	001h	001h
O_RDWR	чтение и запись	002h	002h
O_CREAT	разрешить создание файла	040h	200h
O_EXCL	потребовать создание файла	080h	800h
O_TRUNC	если файл существует, уничтожить его содержимое	200h	400h
O_APPEND	если файл существует, дописывать в конец	400h	008h

Таблица 4.1. Некоторые флаги для второго параметра вызова open

димо проверить, не содержит ли регистр EAX отрицательное значение (в ОС Linux) или не взведён ли флаг CF (в ОС FreeBSD). Если вызов закончился успешно, то регистр EAX содержит ***дескриптор открытого файла*** (потока ввода или вывода). Именно этот дескриптор теперь следует использовать в качестве первого параметра в вызовах read и write для работы с файлом. Как правило, это значение следует сразу же после вызова скопировать в специально отведённую для него область памяти.

Когда все действия с файлом завершены, его следует закрыть. Это делается с помощью вызова close, имеющего номер 6. Вызов принимает один параметр, равный дескриптору закрываемого файла. После этого поток ввода-вывода с таким дескриптором перестаёт существовать; последующие вызовы open могут снова использовать тот же номер дескриптора.

Задача в ОС Unix может узнать свой номер (так называемый идентификатор процесса) с помощью вызова getpid, а также номер своего непосредственного «предка» (процесса, создавшего данный процесс) с помощью вызова getppid. Вызов getpid в обеих рассматриваемых системах имеет номер 20, тогда как вызов getppid имеет номер 64 в ОС Linux и номер 39 в ОС FreeBSD. Оба вызова не принимают параметров; запрашиваемый номер возвращается в качестве результата работы вызова через регистр EAX. Отметим, что эти два вызова всегда завершаются успешно, ошибкам тут просто неоткуда взяться.

Системный вызов kill (номер 37) позволяет отправить сигнал процессу с заданным номером. Вызов принимает два параметра, первый задаёт номер процесса[7], второй задаёт номер сигнала; в частности, сигнал №15 (SIGTERM) предписывает процессу завершиться (но процесс может

[7]На самом деле можно отправить сигнал сразу группе процессов или даже всем процессам в системе, но подробное описание этого выходит за рамки нашего курса.

этот сигнал перехватить и завершиться не сразу, либо вообще не завершаться), а сигнал № 9 (SIGKILL) уничтожает процесс, причём этот сигнал нельзя ни перехватить, ни игнорировать.

Ядра операционных систем семейства Unix поддерживают сотни разнообразных системных вызовов; заинтересованные читатели могут найти информацию об этих вызовах в сети Интернет или в специальной литературе. Отметим, что для ознакомления с информацией о системных вызовах желательно знать язык программирования Си, да и работа на уровне системных вызовов с помощью языка Си строится гораздо проще. Более того, некоторые системные вызовы в отдельных системах могут не поддерживаться ядром, а вместо этого эмулироваться библиотечными функциями Си, что делает их использование в программах на языке ассемблера практически невозможным. В этой связи нелишним будет напомнить, что язык ассемблера мы рассматриваем с учебной, а не практической целью. Программы, предназначенные для практического применения, лучше писать на Си или на других подходящих языках высокого уровня.

§ 4.4. Параметры командной строки

При работе в операционной среде ОС Unix мы, как правило, запускаем программы, указывая кроме их имён ещё и определённые параметры — имена файлов, опции и т. п. Так, при запуске ассемблера NASM мы можем написать что-то вроде

```
nasm -f elf prog.asm
```

Слова, указанные после имени программы, называются ***параметрами командной строки***. В данном случае этих аргументов три: ключ «-f», слово «elf», обозначающее нужный нам формат результата трансляции, и имя файла «prog.asm». Отметим, что и само имя программы, в данном случае «nasm», считается элементом командной строки. Иначе говоря, командная строка представляет собой массив строк, состоящий в данном случае из четырёх элементов: «nasm», «-f», «elf» и «prog.asm».

Естественно, мы и сами можем написать программу, получающую те или иные сведения через командную строку. При запуске программы операционная система отводит в её адресном пространстве специальную область памяти, в которой располагает строки, составляющие командную строку. Информация об адресах этих строк вместе с их общим количеством для удобства помещается в стек запускаемой задачи, после чего управление передаётся нашей программе. Таким образом, в тот момент, когда наша программа начинает выполняться с метки _start, на вершине стека (то есть по адресу [esp]) располагается четырёхбайтное целое число, равное количеству элементов командной строки (включая имя

программы), в следующей позиции стека (по адресу [esp+4]) располагается адрес в памяти, где находится имя, по которому нашу программу вызвали, далее (по адресу [esp+8]) находится адрес первого параметра, потом второго параметра и т. д. Каждый элемент командной строки хранится в памяти в виде строки (массива символов), ограниченной справа нулевым байтом.

Для примера рассмотрим программу, печатающую параметры своей командной строки (включая нулевой). Пользоваться средствами stud_io.inc мы уже не станем, поскольку знаем, как без них обойтись. Для использования вызова write нам понадобится знать длину каждой печатаемой строки, поэтому для удобства мы опишем подпрограмму strlen, получающую в качестве параметра через стек адрес строки и возвращающую через регистр EAX длину этой строки (предполагается, что конец строки обозначен нулевым байтом). Кроме того, отдельную подпрограмму (newline) опишем для печати символа перевода строки; при этом нам потребуется область памяти из одного байта, равного 10, то есть коду перевода строки, чтобы передавать её адрес вызову write, и мы эту область памяти отведём прямо в секции .text[8] вместе с кодом подпрограммы newline сразу после команды ret, пометив локальной меткой.

Ещё один своеобразный момент состоит в том, что наша программа будет расчитана как для работы с ОС Linux, так и для работы с ОС FreeBSD. Поскольку системные вызовы в этих ОС выполняются по-разному, мы воспользуемся директивами условной компиляции для выбора того или иного текста. Эти директивы будут предполагать, что при компиляции под ОС Linux мы определяем (в командной строке NASM) макросимвол OS_LINUX, а при работе под FreeBSD — символ OS_FREEBSD. Таким образом, при работе под ОС Linux наш пример (назовём его cmdl.asm) нужно будет компилировать с помощью команды

```
nasm -f elf -dOS_LINUX cmdl.asm
```

а при работе под ОС FreeBSD — командой

```
nasm -f elf -dOS_FREEBSD cmdl.asm
```

Итак, пишем текст:

```
section .text
global  _start

strlen:         ; arg1 == address of the string
        push ebp
        mov ebp, esp
```

[8]Мы можем так поступить, поскольку эту область памяти наша программа не меняет; если бы это было не так, пришлось бы располагать её в секции .data.

```
        push esi
        xor eax, eax
        mov esi, [ebp+8]    ; arg1
.lp:    cmp byte [esi], 0
        jz .quit
        inc esi
        inc eax
        jmp short .lp
.quit:  pop esi
        pop ebp
        ret

newline:
        pushad
%ifdef OS_FREEBSD
        push dword 1
        push dword .nwl
        push dword 1 ; stdout
        mov eax, 4 ; write
        push eax
        int 80h
        add esp, 16
%elifdef OS_LINUX
        mov edx, 1
        mov ecx, .nwl
        mov ebx, 1
        mov eax, 4
        int 80h
%else
%error please define either OS_FREEBSD or OS_LINUX
%endif
        popad
        ret
.nwl    db 10

_start:
        mov ecx, [esp]
        mov esi, esp
        add esi, 4
again:  push dword [esi]
        call strlen
        add esp, 4
        push esi
        push ecx
%ifdef OS_FREEBSD
        push eax
        push dword [esi]
```

```
        push dword 1 ; stdout
        mov eax, 4 ; write
        push eax
        int 80h
        add esp, 16
%else
        mov edx, eax
        mov ecx, [esi]
        mov ebx, 1
        mov eax, 4
        int 80h
%endif
        call newline
        pop ecx
        pop esi
        add esi, 4
        loop again

%ifdef OS_FREEBSD
        push dword 0
        mov eax, 1 ; _exit
        push eax
        int 80h
%else
        mov ebx, 0
        mov eax, 1
        int 80h
%endif
```

§ 4.5. Пример: копирование файла

Рассмотрим ещё один пример программы, активно взаимодействующей с операционной системой. Эта программа будет получать через параметры командной строки имена двух файлов — оригинала и копии и создавать копию под заданным именем с заданного оригинала. Наша программа будет работать достаточно просто: проверив, что ей действительно передано два параметра, она попытается открыть первый файл на чтение, второй файл — на запись и, если ей это удалось, то циклически читать из первого файла данные порциями по 4096 байт, пока не возникнет ситуация «конец файла». Сразу после чтения каждой порции программа будет записывать прочитанное во второй файл. Настоящая команда `cp`, предназначенная для копирования файлов, устроена гораздо сложнее, но для нашего учебного примера лишняя сложность не нужна.

Ясно, что нашей программе предстоит активно пользоваться системными вызовами. Дело осложняется тем, что нам хотелось бы, конечно,

написать программу, которая будет успешно компилироваться и работать как под ОС Linux, так и под ОС FreeBSD. Как мы видели на примере программы из предыдущего параграфа, это требует довольно громоздкого оформления каждого системного вызова директивами условной компиляции. Предыдущий пример, содержащий всего три системных вызова, можно было написать, не особенно задумываясь над этой проблемой, что мы и сделали; иное дело — программа, в которой предполагается больше десятка обращений к операционной системе. Чтобы не допустить загромождения нашего исходного кода однообразными, но при этом объёмными (и, значит, отвлекающими внимание) конструкциями, мы напишем один многострочный макрос, который и будет осуществлять системный вызов (точнее, он будет генерировать ассемблерный код для осуществления системного вызова). В тексте этого макроса и будут заключены все различия в организации системных вызовов для Linux и FreeBSD. Макрос будет принимать на вход произвольное количество параметров, не меньшее одного; первый параметр будет задавать номер системного вызова, остальные — значения параметров системного вызова. Отметим, что для ОС Linux наш макрос откажется работать с более чем пятью параметрами, поскольку они уже не уместятся в регистры; для FreeBSD такого ограничения нет.

При передаче параметров в макрос и раскладывании их по соответствующим регистрам (в варианте для Linux) мы применим приём, который уже встречали (см. комментарий на стр. 118) — занесение всех параметров в стек с последующим их извлечением в нужные регистры. В варианте для FreeBSD никакого раскладывания по регистрам нам не требуется, зато требуется занести параметры в стек уже для использования их самим системным вызовом. Таким образом, в обоих случаях тело макроса можно начать с занесения в стек всех его параметров (в обратном порядке, чтобы не пришлось их как-либо переупорядочивать в варианте для FreeBSD). Для этого мы воспользуемся директивой `%rotate` точно так же, как мы это уже делали при написании макроса `pcall` (см. стр. 120).

После этого в варианте для FreeBSD достаточно занести номер вызова в `EAX`, и можно инициировать прерывание; в варианте для Linux всё не так просто, нужно ещё извлечь из стека параметры и расположить их в регистрах, причём для различного количества параметров будут задействоваться различные наборы регистров; чтобы корректно обработать всё это, нам придётся написать целый ряд вложенных друг в друга директив условной компиляции, срабатывающих в зависимости от количества переданных макросу параметров.

После возврата из системного вызова наши действия также различаются в зависимости от используемой операционной системы. В случае ОС Linux результат вызова находится в регистре `EAX`, отрицательное зна-

чение указывает на возникшую ошибку, в стеке ничего лишнего нет. В случае ОС FreeBSD на ошибку указывает взведённый флаг CF, в регистре EAX может находиться как результат, так и код ошибки, а в стеке всё ещё лежат параметры вызова, так что стек нуждается в очистке. Мы поступим следующим образом: в случае ОС Linux оставим всё как есть, в случае же ОС FreeBSD проверим флаг CF, и если он взведён, изменим знак регистра EAX на противоположный с помощью команды neg. Таким образом, на выходе мы, как и для ОС Linux, будем иметь в EAX неотрицательное значение в случае успеха и отрицательное — в случае ошибки; после этого мы совершенно спокойно можем испортить содержимое регистра флагов, что, кстати, и произойдёт на следующей команде — мы очистим стек от ненужных уже параметров обычной командой add, которая, как известно, выставляет флаги (включая CF) уже в соответствии со своим результатом.

Окончательно наш макрос будет выглядеть так:

```
%macro        syscall 1-*
%rep %0
%rotate -1
        push dword %1
%endrep
%ifdef OS_FREEBSD
        mov eax, [esp]
        int 80h
        jnc %%sc_ok
        neg eax
%%sc_ok:
        add esp, (%0-1)*4
%elifdef OS_LINUX
        pop eax
 %if %0 > 1
        pop ebx
  %if %0 > 2
        pop ecx
   %if %0 > 3
        pop edx
    %if %0 > 4
        pop esi
     %if %0 > 5
        pop edi
      %if %0 > 6
       %error "Too many params for Linux syscall"
      %endif
     %endif
    %endif
   %endif
```

```
  %endif
 %endif
        int 80h
%else
%error Please define either OS_LINUX or OS_FREEBSD
%endif
%endmacro
```

Текст макроса, конечно, получился достаточно длинным, но это компенсируется сокращением объёма основного кода. Так, рассказывая о конвенциях системных вызовов, мы привели код программы, печатающей одну строку, в варианте для Linux (стр. 137) и FreeBSD (стр. 139). С использованием вышеприведённого макроса мы можем написать так:

```
section .data
msg     db "Hello world", 10
msg_len equ $-msg
section .text
global  _start
_start: syscall 4, 1, msg, msg_len
        syscall 1, 0
```

и всё, причём эта программа будет компилироваться и правильно работать под обеими системами, нужно только не забывать указывать NASM'у флаг `-dOS_LINUX` или `-dOS_FREEBSD`.

Вернёмся к нашей задаче копирования. В программе нам потребуется буфер для временного хранения данных, в который мы будем считывать очередную порцию данных из первого файла, чтобы затем записать её во второй файл. Кроме того, нам будут нужны переменные для хранения дескрипторов файлов (хранить их в регистрах будет сложно, ведь каждый системный вызов может испортить значения регистров); соответствующие переменные мы назовём `fdsrc` и `fddest`. Наконец, мы для удобства заведём переменные для хранения количества параметров командной строки и адреса начала массива указателей на параметры командной строки, назвав эти переменные `argc` и `argvp`. Все эти переменные не требуют начальных значений и могут, таким образом, быть расположены в секции `.bss`:

```
section .bss
buffer  resb    4096
bufsize equ     $-buffer
fdsrc   resd    1
fddest  resd    1
argc    resd    1
argvp   resd    1
```

Наша программа может обнаружить одну из трёх ошибок: пользователь может указать неправильное количество параметров командной строки,

может указать несуществующий или недоступный файл в качестве источника данных, либо может указать в качестве целевого такой файл, который мы по каким-то причинам не сможем открыть на запись. В первом случае пользователю следует объяснить, с какими параметрами следует запускать нашу программу, в остальных двух — просто сообщить о происшедшей ошибке. Все три сообщения об ошибках мы расположим в секции `.data` в виде инициализированных переменных:

```
section .data
helpmsg db 'Usage: copy <src> <dest>', 10
helplen equ $-helpmsg
err1msg db "Couldn't open source file for reading", 10
err1len equ $-err1msg
err2msg db "Couldn't open destination file for writing", 10
err2len equ $-err1msg
```

Теперь мы можем приступить к написанию секции `.text`, то есть самой программы, и в самом начале мы проверим, что нам передано ровно два параметра. Для этого мы извлечём из стека лежащее на его вершине число, обозначающее количество элементов командной строки, занесём его в переменную `argc`. Заодно на всякий случай сохраним адрес текущей вершины стека в переменной `argvp`, но извлекать из стека больше ничего не будем, так что в области стека у нас окажется массив адресов строк–элементов командной строки. Проверим, что в переменной `argc` оказалось число 3; правильная командная строка должна в нашем случае состоять из трёх элементов: имени самой программы и двух параметров. В случае, если количество параметров окажется неверным, напечатаем пользователю сообщение об ошибке и выйдем:

```
section .text
global  _start
_start:
        pop dword [argc]
        mov [argvp], esp
        cmp dword [argc], 3
        je .args_count_ok
        syscall 4, 2, helpmsg, helplen
        syscall 1, 1
.args_count_ok:
```

Следующим нашим действием должно стать открытие файла, имя которого задано первым параметром командной строки, на чтение. Мы помним, что в переменной `argvp` находится адрес в памяти (стековой), начиная с которого располагаются адреса элементов командной строки. Извлечём адрес из `argvp` в регистр ESI, затем возьмём четырёхбайтное

значение по адресу [esi+4] — это и будет адрес первого параметра командной строки, то есть строки, задающей имя файла, который надо читать и копировать. Для хранения адреса воспользуемся регистром EDI, после чего сделаем вызов open. Нам придётся использовать два параметра — собственно адрес имени файла и режим его использования, который будет в данном случае равен 0 (O_RDONLY). Результат работы системного вызова нам обязательно надо будет проверить; напомним, что наш макрос syscall устроен так, чтобы отрицательное значение EAX указывало на ошибку, а неотрицательное — на успешное выполнение вызова; в применении к вызову open результатом успешного его выполнения является дескриптор нового потока ввода-вывода, в данном случае это поток ввода, связанный с копируемым файлом. В случае успеха сохраним полученный дескриптор в переменной fdsrc, в случае неудачи — выдадим сообщение об ошибке и выйдем.

```
        mov esi, [argvp]
        mov edi, [esi+4]
        syscall 5, edi, 0 ; O_RDONLY
        cmp eax, 0
        jge .source_open_ok
        syscall 4, 2, err1msg, err1len
        syscall 1, 2
.source_open_ok:
        mov [fdsrc], eax
```

Настало время открыть второй файл на запись. Для извлечения его имени из памяти воспользуемся точно так же регистрами ESI и EDI, после чего выполним системный вызов open, в случае ошибки выдадим сообщение и выйдем, в случае успеха сохраним дескриптор в переменной fddest. Вызов open в этот раз будет несколько сложнее. Во-первых, режим открытия на этот раз задаётся флажками O_WRONLY, O_CREAT и O_TRUNC, два из которых, как это обсуждалось на стр. 141, имеют различные числовые значения в ОС Linux и ОС FreeBSD. Во-вторых, поскольку в этот раз возможно создание нового файла, наш системный вызов должен получить ещё и третий параметр, который, как мы ранее отмечали, обычно равен 666o. С учётом всего этого получится такой код:

```
        mov esi, [argvp]
        mov edi, [esi+8]
%ifdef OS_LINUX
        syscall 5, edi, 241h, 0666o
%else           ; assume it's FreeBSD
        syscall 5, edi, 601h, 0666o
%endif
        cmp eax, 0
```

```
        jge .dest_open_ok
        syscall 4, 2, err2msg, err2len
        syscall 1, 3
.dest_open_ok:
        mov [fddest], eax
```

Наконец, напишем основной цикл. В нём мы будем выполнять чтение из первого файла, анализировать его результат, и если достигнут конец файла (в `EAX` значение `0`) или произошла ошибка (отрицательное значение), то будем выходить из цикла, ну а если чтение прошло успешно, то нужно будет записать всё прочитанное (то есть столько байтов из области память `buffer`, какое число содержится в `EAX`) во второй файл.

```
.again: syscall 3, [fdsrc], buffer, bufsize
        cmp eax, 0
        jle .end_of_file
        syscall 4, [fddest], buffer, eax
        jmp .again
```

Выход из цикла мы производили переходом на метку `end_of_file`; рано или поздно наша программа, достигнув конца первого файла, перейдёт на эту метку, после чего нам останется только закрыть оба файла вызовом `close` и завершить программу:

```
.end_of_file:
        syscall 6, [fdsrc]
        syscall 6, [fddest]
        syscall 1, 0
```

Отметим, что все метки в основной программе, кроме метки `_start`, мы сделали локальными (их имена начинаются с точки). Так делать не обязательно, но такой подход к меткам (все метки, к которым не предполагается обращаться откуда-то издалека, делать локальными) позволяет в более крупных программах избежать проблем с конфликтами имён.

Глава 5. Раздельная трансляция

§ 5.1. Что такое модули и зачем они нужны

До сих пор все программы, которые мы писали на языке ассемблера, умещались в одном файле. Иногда мы использовали несколько файлов, но соединение их воедино производилось на этапе макропроцессирования, то есть ещё до начала перевода программы в машинный код.

Пока исходный текст программы состоит из нескольких десятков строк, его действительно удобнее всего хранить в одном файле. С увеличением объема программы, однако, работать с одним файлом становится всё труднее и труднее, и тому можно назвать несколько причин. Во-первых, длинный файл элементарно тяжело перелистывать. Во-вторых, как правило, программист в каждый момент времени работает только с небольшим фрагментом исходного кода, старательно выкидывая из головы остальные части программы, чтобы не отвлекаться, и в этом плане было бы лучше, чтобы фрагменты, не находящиеся в работе в настоящий момент, располагались бы где-нибудь подальше, то есть так, чтобы не попадаться на глаза программисту даже случайно. В-третьих, если программа разбита на отдельные файлы, в ней оказывается гораздо проще найти нужное место, подобно тому, как проще найти нужную бумагу в шкафу с офисными папками, нежели в большом ящике безо всяких папок. Наконец, часто бывает так, что один и тот же фрагмент кода используется в разных программах — а ведь его, скорее всего, так или иначе приходится время от времени редактировать, чтобы, например, исправить ошибки, и тут уже совершенно очевидно, что гораздо проще исправить файл в одном месте и скопировать (файл целиком) во все остальные проекты, чем исправлять один и тот же фрагмент, который вставлен в разные файлы.

Разбивка текста программы на файлы, соединяемые директивами `%include` или их аналогами, снимает часть проблем, но, к сожалению,

не все, поскольку такой набор файлов остаётся, как говорят программисты, *одной единицей трансляции* — иначе говоря, мы можем их транслировать с помощью ассемблера (или с помощью компилятора, если мы пишем на языке высокого уровня) только все вместе, за один приём. Прежде всего тут возникает проблема со скоростью трансляции. Современные компиляторы и ассемблеры работают довольно быстро, но объемы наиболее серьёзных программ таковы, что их полная перекомпиляция может занять несколько часов, а иногда и несколько суток. Если после внесения любого, даже самого незначительного изменения в программу нам, чтобы посмотреть, что получилось, придётся ждать сутки (да и пару часов — этого уже будет достаточно) — работать станет совершенно невозможно. Более того, программисты практически всегда используют так называемые ***библиотеки*** — комплекты готовых подпрограмм, которые почти никогда не изменяются и, соответственно, постоянно тратить время на их перекомпиляцию было бы несколько глупо. Наконец, проблемы создают и постоянно возникающие конфликты имён: чем больше объём кода, тем больше в нём требуется меток и других идентификаторов, растёт вероятность случайных совпадений, а сделать с этим при трансляции в один приём почти ничего нельзя — ведь даже локальные метки, как мы уже говорили, на самом деле представляют собой не более чем укороченную запись более длинных глобальных меток.

Все эти проблемы позволяет решить техника ***раздельной компиляции***. Суть её в том, что программа создаётся в виде множества обособленных частей, каждая из которых транслируется отдельно. Такие части называются ***единицами трансляции*** или ***модулями***. Чаще всего в роли модулей выступают отдельные файлы. Обычно в виде обособленной единицы трансляции оформляют набор логически связанных между собой подпрограмм; в модуль также помещают и всё необходимое для их работы — например, глобальные переменные, если такие есть, а также всевозможные константы и прочее. Каждый модуль транслируется отдельно; в результате трансляции каждого из них получается ***объектный файл***, обычно имеющий суффикс «`.o`». Затем с помощью редактора связей из набора объектных файлов получают исполняемый файл.

Очень важным свойством модуля является наличие у него собственного ***пространства имён***. Метки, введённые в модуле, будут видны только из других мест того же модуля, если только мы специально не объявим их «глобальными» (напомним, что в языке ассемблера NASM это делается директивой `global`). Часто бывает так, что модуль вводит несколько десятков, а иногда и сотен меток, но все они оказываются нужны только в нём самом, а из всей остальной программы требуются обращения лишь к одной-двум процедурам. Это практически снимает проблему конфликтов имён: в разных модулях могут появляться метки с одинаковыми именами, и это никак нам не мешает, если только они не

глобальные. Технически это означает, что при трансляции исходного текста модуля в объектный код все метки, кроме объявленных глобальными, исчезают, так что в объектном файле содержится уже только информация об именах глобальных меток.

Интересно, что собственные пространства имён модулей позволяют решить не только проблему конфликта имён, но и проблему простейшей «защиты от дурака», особенно актуальной в крупных программных разработках, в которых принимает участие несколько человек. Если автор модуля не предполагает, что та или иная процедура будет вызываться из других модулей, либо что переменная не должна изменяться никак иначе, чем процедурами того же модуля, то ему достаточно не объявлять соответствующие метки глобальными, и можно ни о чём не беспокоиться — обратиться к ним другие программисты не смогут чисто технически. Такое сокрытие деталей реализации той или иной подсистемы в программе называется ***инкапсуляцией*** и позволяет, например, более смело исправлять код модулей, не боясь, что другие модули при этом перестанут работать: достаточно сохранять неизменными и работающими глобальные метки.

§ 5.2. Поддержка модулей в NASM

Ассемблер NASM поддерживает модульное программирование, вводя для этого два основных понятия: ***глобальные метки*** и ***внешние метки***. С первыми из них мы уже знакомы: такие метки объявляются директивой `global` и, как мы уже знаем, отличаются от обычных тем, что информация о них включается в объектный файл модуля и становится, таким образом, видна системному редактору связей. Что касается внешних меток, то это, напротив, метки, ***введения которых мы ожидаем от других модулей***. Чаще всего это просто имя подпрограммы (реже — глобальной переменной), которая описана где-то в другом модуле, но к которой нам необходимо обратиться. Чтобы это стало возможным, необходимо сообщить ассемблеру о существовании этой метки. Действительно, ассемблер во время трансляции видит только текст одного модуля и ничего не знает о том, что в других модулях объявлены те или иные метки, так что, если мы попытаемся обратиться к метке из другого модуля, никак не сообщив ассемблеру о факте её существования, мы попросту получим сообщение об ошибке. Для этого ассемблер NASM вводит директиву `extern`. Например, если мы пишем модуль, в котором хотим обратиться к процедуре `myproc`, а сама эта процедура описана где-то в другом месте, то, чтобы сообщить об этом, следует написать:

```
extern myproc
```

Такая строка приказывает ассемблеру буквально следующее: «метка `myproc` существует, хотя её и нет в текущем модуле, так что, встретив такую метку, просто сгенерируй соответствующий объектный код, а конкретный адрес вместо этой метки потом подставит редактор связей».

§ 5.3. Пример

В качестве примера многомодульной программы мы напишем простую программу, которая спрашивает у пользователя его имя, а затем здоровается с ним по имени. Работу со строками мы на этот раз организуем так, как это обычно делается в программах на языке Си: будем использовать нулевой байт в качестве признака конца строки. Головная программа будет зависеть от двух основных подпрограмм, `putstr` и `getstr`, каждую из которых мы вынесем в отдельный модуль. Подпрограмме `putstr` потребуется посчитать длину строки, чтобы напечатать всю строку за одно обращение к операционной системе; для такого подсчёта мы используем функцию `strlen`, уже знакомую нам по программе из § 4.4. Её мы тоже вынесем в отдельный модуль. Наконец, организацию вызова `_exit` мы тоже вынесем в подпрограмму (назовём её `quit`) и в отдельный модуль. Все модули назовём так же, как и вынесенные в них подпрограммы: `putstr.asm`, `getstr.asm`, `strlen.asm` и `quit.asm`.

Для организации системных вызовов мы используем макрос `syscall`, который мы описали на стр. 148. Его мы также вынесем в отдельный файл, но полноценным модулем этот файл быть не сможет. Действительно, модуль — это единица трансляции, тогда как макрос, вообще говоря, не может быть ни во что оттранслирован: как мы отмечали ранее, в ходе трансляции макросы полностью исчезают и в объектном коде от них ничего не остаётся. Это и понятно, ведь макросы представляют собой набор указаний не для процессора, а для самого ассемблера, и чтобы от макроса была какая-то польза, ассемблер должен, разумеется, видеть определение макроса в том месте, где он встретит обращение к этому макросу. Поэтому файл, содержащий наш макрос `syscall`, мы будем подсоединять к другим файлам с помощью директивы `%include` на стадии препроцессирования (в отличие от модулей, которые собираются в единое целое существенно позже — после завершения трансляции, с помощью редактора связей). Этот файл мы назовём `syscall.inc`; с него мы вполне можем начать, открыв его для редактирования и набрав в нём ровно такое определение макроса, какое было дано на стр. 148; ничего другого в этом файле набирать не требуется.

Следующим мы напишем файл `strlen.asm`. Он будет выглядеть так:

```
global  strlen

section .text
; procedure strlen
; [ebp+8] == address of the string
strlen: push ebp
        mov ebp, esp
        xor eax, eax
```

```
        mov esi, [ebp+8]
.lp:    cmp byte [esi], 0
        jz .quit
        inc esi
        inc eax
        jmp short .lp
.quit:  pop ebp
        ret
```

Первая строчка файла указывает, что в этом модуле будет определена метка `strlen` и эту метку необходимо сделать видимой из других модулей. Вообще говоря, мы могли бы поставить эту директиву где угодно, но лучше вынести её в начало, чтобы при первом же взгляде на текст модуля можно было догадаться, для чего он нужен. Подробно комментировать текст процедуры мы не будем, поскольку он нам уже знаком.

Имея в своём распоряжении процедуру `strlen`, напишем модуль `putstr.asm`. Процедура `putstr` будет вызывать `strlen` для подсчёта длины строки, а затем обращаться к системному вызову `write`:

```
%include "syscall.inc"      ; нужен макрос syscall
global  putstr              ; модуль описывает putstr
extern  strlen              ; а сам использует strlen

section         .text
; procedire putstr
; [ebp+8] = address of the string
putstr: push ebp            ; стандартное начало
        mov ebp, esp        ;   подпрограммы
        push dword [ebp+8]  ; вызываем strlen для
        call strlen         ;   подсчёта длины строки
        add esp, 4          ; результат теперь в EAX
        syscall 4, 1, [ebp+8], eax  ; вызываем write
        mov esp, ebp        ; стандартное завершение
        pop ebp             ;   подпрограммы
        ret
```

Теперь настал черёд самого сложного из модулей нашей программы — модуля `getstr`. Процедура `getstr` будет получать на вход адрес буфера, в котором следует разместить прочитанную строку, и (на всякий случай) длину этого буфера, чтобы не допустить его переполнения, если пользователю придёт в голову набрать строку, которая в наш буфер не поместится. Для упрощения реализации мы будем считывать строку по одному символу; конечно, в настоящих программах так не делают, но наша задача сейчас не в том, чтобы получить эффективную программу, так что мы вполне можем немного облегчить себе жизнь. Подпрограмма

getstr будет использовать локальную переменную, которую в комментариях мы назовём I и которая, как и все локальные переменные, будет располагаться в стековом фрейме, для чего мы в начале процедуры соответствующим образом изменим указатель стека. В переменной I будет содержаться *текущее количество прочитанных символов*, изначально равное нулю. Далее процедура будет в цикле читать по одному символу с помощью системного вызова read. Чтение будет прекращено при наступлении одного из следующих условий: либо read вернёт что-либо отличное от 1, что в данном случае будет означать наступление ситуации «конец файла» или ошибку; либо код прочитанного символа будет равен 10, то есть это окажется символ перевода строки (этот код генерирует клавиша Enter); либо, наконец, в буфере останется место только под завершающий нулевой байт, что проверяется условием I+1⩾buflen. После выхода из цикла а конец буфера записывается ограничительный нулевой байт. В случае, если причиной выхода из цикла был прочитанный код символа перевода строки, нулевой байт записывается на его место, чтобы в буфере никаких переводов строки не содержалось; это достигается уменьшением переменной I перед выходом из цикла.

Полностью текст модуля getstr.asm будет выглядеть так:

```
%include "syscall.inc"       ; нужен макрос syscall
global getstr                ; модуль описывает getstr

section .text
; procedure getstr
; [ebp+8]  = address of buffer
; [ebp+12] = length of buffer
getstr: push ebp             ; стандартное начало
        mov ebp, esp         ;   подпрограммы
        sub esp, 4           ; место под переменную I
        xor eax, eax         ; eax:=0
        mov [ebp-4], eax     ; I:=0
.again:                      ; начало главного цикла
        mov eax, [ebp+8]     ; заносим адрес в EAX
        add eax, [ebp-4]     ; прибавляем к нему I
        syscall 3, 0, eax, 1 ; вызываем read
        cmp eax, 1           ; вернул ли он 1?
        jne .eol             ; нет - выйти из цикла
        mov eax, [ebp+8]     ; заносим адрес в EAX
        add eax, [ebp-4]     ; прибавляем к нему I
        mov bl, [eax]        ; считанный байт (в BL)
        cmp bl, 10           ; равен 10?
        jne .noeol           ; нет - перепрыгиваем
        dec dword [ebp-4]    ; да - уменьшаем I
        jmp .eol             ;   и выходим из цикла
.noeol: mov eax, [ebp-4]     ; загружаем I
```

```
        inc eax                 ; теперь в EAX зн. I+1
        cmp eax, [ebp+12]       ; не превышает ли arg2?
        jae .eol                ; да - выходим из цикла
        inc dword [ebp-4]       ; увеличиваем I
        jmp .again              ; продолжаем цикл
.eol:   mov eax, [ebp+8]        ; загружаем адрес в EAX
        add eax, [ebp-4]        ; прибавляем I
        inc eax                 ; прибавляем 1
        xor bl, bl              ; обнуляем BL
        mov [eax], bl           ; заносим 0 в конец строки
        mov esp, ebp            ; стандартный выход
        pop ebp                 ;   из подпрограммы
        ret
```

Напишем теперь самый простой из наших модулей — `quit.asm`:

```
%include "syscall.inc"
global  quit
section .text
quit:   syscall 1, 0
```

Все подпрограммы готовы, и мы можем приступать к написанию головного модуля, который мы назовём `greet.asm`. Поскольку все обращения к системным вызовам мы вынесли в подпрограммы, в головном модуле макрос `syscall` (а, значит, и включение файла `syscall.inc`) нам не понадобится. Текст выдаваемых программой сообщений мы опишем, как обычно, в виде инициализированных строк в секции `.data`; надо только не забывать, что в этой программе все строки должны иметь ограничивающий их нулевой байт. Буфер для чтения строки мы разместим в секции `.bss`. Что касается секции `.text`, то она будет состоять из сплошных вызовов подпрограмм.

```
global  _start                  ; это головной модуль
extern  putstr                  ; он использует подпрограммы
extern  getstr                  ;   putstr, getstr и quit
extern  quit

section .data                   ; описываем текст сообщений
nmq     db      'Hi, what is your name?', 10, 0
pmy     db      'Pleased to meet you, dear ', 0
exc     db      '!', 10, 0

section .bss                    ; выделяем память под буфер
buf     resb    512
buflen  equ     $-buf

section .text
```

```
_start: push dword nmq      ; начало головной программы
        call putstr         ; вызываем putstr для nmq
        add esp, 4
        push dword buflen   ; вызываем getstr
        push dword buf      ;   с параметрами buf и
        call getstr         ;   buflen
        add esp, 8
        push dword pmy      ; вызываем putstr для pmy
        call putstr
        add esp, 4
        push dword buf      ; вызываем putstr для
        call putstr         ;   строки, введённой
        add esp, 4          ;   пользователем
        push dword exc      ; вызываем putstr для exc
        call putstr
        add esp, 4
        call quit           ; вызываем quit
```

Итак, в нашей рабочей директории теперь находятся файлы `syscall.inc`, `strlen.asm`, `putstr.asm`, `getstr.asm`, `quit.asm` и `greet.asm`. Чтобы получить рабочую программу, нам понадобится отдельно вызвать NASM для каждого из модулей (напомним, что `syscall.inc` модулем не является):

```
nasm -f elf -dOS_LINUX strlen.asm
nasm -f elf -dOS_LINUX putstr.asm
nasm -f elf -dOS_LINUX getstr.asm
nasm -f elf -dOS_LINUX quit.asm
nasm -f elf -dOS_LINUX greet.asm
```

Отметим, что флажок `-dOS_LINUX` необходим только для тех модулей, которые используют `syscall.inc`, так что мы могли бы при компиляции `strlen.asm` и `greet.asm` его не указывать. Однако практика показывает, что проще указывать такие флажки всегда, нежели чем помнить, для каких модулей они нужны, а для каких — нет.

Результатом работы NASM станут пять файлов с суффиксом «`.o`», представляющие собой *объектные модули* нашей программы. Чтобы объединить их в исполняемый файл, мы вызовем редактор связей `ld`:

```
ld greet.o strlen.o getstr.o putstr.o quit.o -o greet
```

Результатом на сей раз станет исполняемый файл `greet`, который мы, как обычно, запустим на исполнение командой `./greet`.

§ 5.4. Объектный код и машинный код

Из приведённых выше примеров видно, что каждый объектный модуль, кроме всего прочего, характеризуется списком символов (в терминах ассемблера — меток), которые он предоставляет другим модулям, а также списком символов, которые ему самому должны быть предоставлены другими модулями. Буквально переведя с английского языка названия соответствующих директив (`global` и `extern`), мы можем назвать такие символы «глобальными» и «внешними»; чаще, однако, их называют «экспортируемыми» и «импортируемыми».

Ясно, что при трансляции исходного текста ассемблер, видя обращение к внешней метке, не может заменить эту метку конкретным адресом, поскольку этот адрес ему не известен — ведь метка определена в другом модуле, которого ассемблер не видит. Таким образом, всё, что может сделать ассемблер — это оставить под такой адрес свободное место в итоговом коде и записать в объектный файл информацию, которая позволит редактору связей расставить все такие «пропущенные» адреса, когда их значения уже будут известны. При ближайшем рассмотрении оказывается, что заменить метки конкретными адресами ассемблер не может не только в случае обращений к внешним меткам, но **вообще никогда**. Дело в том, что, коль скоро программа состоит из нескольких (скольки угодно) модулей, ассемблер при трансляции одного из них никак не может предугадать, каким по счёту этот модуль будет стоять в итоговой программе, какого размера будут все предшествующие модули и, таким образом, не может знать, в какой области памяти (даже виртуальной) будет располагаться тот код, который ассемблер в настоящее время генерирует.

С другой стороны, известно, что редактор связей не видит исходных текстов модулей, да и не может их видеть, поскольку предназначен для связи модулей, полученных различными компиляторами из исходных текстов на, вполне возможно, разных языках программирования. Следовательно, вся информация, необходимая для окончательного превращения объектного кода в исполняемый машинный, должна быть записана в объектный файл. Таким образом, объектный код, который получается в качестве результата ассемблирования, представляет собой некий «полуфабрикат» машинного кода, в котором вместо абсолютных (числовых) адресов находится некая информация о том, как эти адреса вычислить и в какие места кода их следует расставить.

Отметим, что информацию о символах, содержащихся в объектном файле, можно узнать с помощью программы `nm`. В качестве упражнения попробуйте применить эту программу к объектным файлам написанных вами модулей (либо модулей из приведённых выше примеров) и попытаться проинтерпретировать результаты.

§ 5.5. Библиотеки

Чаще всего программы пишутся не «с абсолютного нуля», как это в большинстве примеров делали мы, а используют комплекты уже готовых подпрограмм, оформленные в виде *библиотек*. Естественно, такие подпрограммы входят в состав модулей, а сами модули удобнее иметь в заранее откомпилированном виде, чтобы не тратить время на их компиляцию; разумеется, полезно иметь в доступности и исходные тексты этих модулей, но в заранее откомпилированной форме библиотеки используются чаще. Вообще говоря, различают программные библиотеки разных видов; например, бывают библиотеки макросов, которые, естественно, не могут быть заранее откомпилированы и существуют только в виде исходных текстов. Здесь мы, однако, рассмотрим более узкое понятие, а именно то, что под термином «библиотека» понимается на уровне редактора связей.

С технической точки зрения библиотека подпрограмм — это файл, объединяющий в себе некоторое количество объектных модулей и, как правило, содержащий таблицы для ускоренного поиска имён символов в этих модулях.

Необходимо отметить одно важнейшее свойство объектных файлов: каждый из них может быть включён в итоговую программу **только целиком** либо не включён вообще. Это означает, например, что если вы объединили в одном модуле несколько подпрограмм, а кому-то потребовалась лишь одна из них, в исполняемый файл всё равно войдёт код всего вашего модуля (то есть всех подпрограмм). Это необходимо учитывать при разбиении библиотеки на модули; так, системные библиотеки, поставляемые вместе с операционными системами, компиляторами и т. п., обычно строятся по принципу «одна функция — один модуль».

Для построения библиотеки из отдельных объектных модулей необходимо использовать специально предназначенные для этого программы. В ОС Unix соответствующая программа называется `ar`. Изначально её предназначение не ограничивалось созданием библиотек (само название `ar` означает «архиватор»), так что при вызове программы необходимо указать с помощью параметра командной строки, чего мы от неё добиваемся. Так, если бы мы захотели объединить в библиотеку все модули программы `greet` (кроме, разумеется, главного модуля, который не может быть использован в других программах), это можно было бы сделать следующей командой:

```
ar crs libgreet.a strlen.o getstr.o putstr.o quit.o
```

Результатом станет файл `libgreet.a`; это и есть библиотека. После этого скомпоновать программу `greet` с помощью редактора связей можно, например, так:

```
ld greet.o libgreet.a
```

или так:

```
ld greet.o -l greet -L .
```

В отличие от монолитного объектного файла, библиотека, будучи упакованной в один файл, продолжает, тем не менее, быть именно *набором объектных модулей*, из которых редактор связей выбирает только те, которые ему нужны для удовлетворения неразрешённых ссылок. Подробнее об этом мы расскажем в следующем параграфе.

§ 5.6. Алгоритм работы редактора связей

Редактору связей в командной строке указывается список объектов, каждый из которых может быть либо объектным файлом, либо библиотекой, при этом объектные файлы могут быть заданы только по имени файла, тогда как библиотеки могут задаваться двумя способами: либо явным указанием имени файла, либо — с помощью флага `-l` — указанием ***имени библиотеки***, которое может упрощённо пониматься как имя файла библиотеки, от которого отброшены префикс `lib` и суффикс `.a`[1]. Так, в примере из предыдущего параграфа файл библиотеки назывался `libgreet.a`, а соответствующее *имя библиотеки* представляло собой слово `greet`. При использовании флага `-l` редактор связей пытается найти файл библиотеки с соответствующим именем в системных директориях (`/lib`, `/usr/lib` и т. п.), но можно указать ему дополнительные директории с помощью флага `-L`; так, «`-L .`» означает, что следует сначала попробовать найти библиотеку в текущей директории, и лишь затем начинать поиск в системных директориях.

В своей работе редактор связей использует два *списка символов*: список известных (разрешённых, от английского *resolved*) символов и список *неразрешённых ссылок (unresolved links)*. В первый список заносятся символы, *экспортируемые* объектными модулями (в своих текстах на языке ассемблера NASM мы помечали такие символы директивой `global`), во второй список заносятся символы, к которым уже есть обращения, то есть имеются модули, *импортирующие* эти символы (для NASM это символы, объявленные директивой `extern` и затем использованные), но которые пока не встретились ни в одном из модулей в качестве экспортируемых.

Редактор связей начинает работу, инициализировав оба списка символов как пустые, и шаг за шагом продвигается слева направо по списку

[1] Мы здесь не рассматриваем случай так называемых разделяемых библиотек, файлы которых имеют суффикс `.so`; концепция динамической загрузки требует дополнительного обсуждения, которое выходит за рамки данного пособия.

объектов, указанных в его командной строке. В случае, если очередным указанным объектом будет объектный файл, редактор связей «принимает» его в формируемый исполняемый файл. При этом все символы, экспортируемые этим модулем, заносятся в список известных символов; если некоторые из них присутствовали в списке неразрешённых ссылок, они оттуда удаляются. Символы, импортируемые модулем, заносятся в список неразрешённых ссылок, если только они к этому времени не фигурируют в списке известных символов. Объектный код из модуля принимается редактором связей к последующему преобразованию в исполняемый код и вставке в исполняемый файл.

Если же очередным объектом из списка, указанного в командной строке, окажется библиотека, действия редактора связей будут более сложными и гибкими, поскольку возможно, что принимать *все* составляющие библиотеку модули ни к чему. Прежде всего редактор связей сверится со списком неразрешённых ссылок; если этот список пуст, библиотека будет полностью проигнорирована как ненужная. Однако обычно список в такой ситуации не пуст (иначе программист не стал бы указывать библиотеку), и следующим действием редактора связей становятся поочерёдные попытки найти в библиотеке такие модули, которые экспортируют один или несколько символов с именами, фигурирующими в текущем списке неразрешённых ссылок; если такой модуль найден, редактор связей «принимает» его, соответствующим образом модифицирует списки символов и начинает рассмотрение библиотеки снова, и так до тех пор, когда ни один из оставшихся в библиотеке непринятых модулей не будет пригоден для разрешения ссылок. Тогда редактор связей прекращает рассмотрение библиотеки и переходит к следующему объекту из списка. Таким образом, из библиотеки берутся только те модули, которые нужны, чтобы удовлетворить потребности предшествующих модулей в импорте символов, плюс, возможно, такие модули, в которых нуждаются уже принятые модули из той же библиотеки. Так, при сборке программы `greet` из предыдущего параграфа редактор связей сначала принял из библиотеки `libgreet.a` модули `getstr`, `putstr` и `quit`, поскольку в них присутствовали символы, импортируемые ранее принятым модулем `greet.o`; затем редактор связей принял и модуль `strlen`, поскольку в нём нуждался модуль `putstr`.

Редактор связей выдаёт сообщения об ошибках и отказывается продолжать сборку исполняемого файла в двух основных случаях. Первый из них возникает, когда список объектов (модулей и библиотек) исчерпан, а список неразрешённых ссылок не опустел, то есть как минимум один из принятых модулей ссылается в качестве внешнего на символ, который так ни в одном из модулей и не встретился; такая ошибочная ситуация называется ***неопределённой ссылкой*** (англ. *undefined reference*). Второй случай ошибочной ситуации — это появление в очередном при-

нимаемом модуле экспортируемого символа, который к этому моменту уже значится в списке известных; иначе говоря, два или более принятых к рассмотрению модуля экспортируют один и тот же символ. Это называется ***конфликтом имён***[2].

Интересно, что **редактор связей никогда не возвращается назад** в своём движении по списку объектов, так что если некоторый модуль из состава библиотеки не был принят на момент, когда редактор до этой библиотеки добрался, то потом он не будет принят тем более, даже если в каком-либо из последующих модулей появится импортируемый символ, который можно было бы разрешить, приняв ещё модули из ранее обработанной библиотеки. Из этого факта вытекает важное следствие: объектные модули следует указывать **раньше**, чем библиотеки, в которых эти модули нуждаются. Вторым важным следствием является то, что **библиотеки никогда не должны «перекрёстно» зависеть друг от друга**, то есть если одна библиотека использует возможности второй, то вторая не должна использовать возможности первой. Если подобного рода перекрёстные зависимости возникли, такие две библиотеки следует объединить в одну.

Наконец, можно сделать ещё один вывод. До тех пор, пока библиотеки вообще не зависят друг от друга, мы можем не слишком волноваться о порядке параметров для редактора связей: достаточно сначала указать в произвольном порядке все объектные файлы, составляющие нашу программу, а затем, опять-таки в произвольном порядке, перечислить все нужные библиотеки. Если же зависимости между библиотеками появляются, порядок их указания становится важен, и при его несоблюдении программа не соберётся. Таким образом, зависимости библиотек друг от друга, даже не перекрёстные, порождают определённые проблемы. Поэтому, прежде чем полагаться при разработке одной библиотеки на возможности другой, следует многократно и тщательно всё обдумать.

Знание принципов работы редактора связей пригодится вам не только (и не столько) в учебном программировании на языке ассемблера, но и в практической работе на языках программирования высокого уровня, в особенности на языках Си и Си++. Не принимая во внимание содержание этого параграфа, вы рискуете, с одной стороны, перегрузить свои исполняемые файлы ненужным (неиспользуемым) содержимым, а с другой — спроектировать свои библиотеки так, что даже сами начнёте в них путаться.

---

[2]Современные редакторы связей в угоду нерадивым программистам позволяют не считать некоторые случаи конфликта имён ошибкой; это используется, например, компиляторами языка Си++. Постарайтесь, насколько возможно, не полагаться на подобные возможности.

Глава 6. Арифметика с плавающей точкой

До сих пор мы рассматривали только целые числа, и лишь вскользь упоминали о существовании альтернативы. Между тем, при выполнении численных расчётов (например, в задачах, связанных с моделированием физических явлений и процессов) целочисленная арифметика оказывается неудобна; можно, конечно, прибегнуть к методу ***фиксированной точки*** (считать, что используемые целые числа представляют не единицы, а, например, десятитысячные доли единиц), но для серьёзных расчётов это не подходит. Альтернативой является работа с машинным представлением дробных чисел в виде двоичных дробей. Такое представление обычно считается *приблизительным*, а в ходе работы при выполнении арифметических операций возникают *ошибки округления*; это неизбежная плата за представление непрерывных (по своей сути) величин дискретным способом.

В ранних процессорах линейки x86 (вплоть до 80386) возможности работы с числами с плавающей точкой отсутствовали; их можно было либо эмулировать программно (работала такая эмуляция очень медленно), либо установить в компьютер дополнительную микросхему, называемую ***арифметическим сопроцессором***: 8087 для 8086, 80287 для 80287, и, наконец, 80387 для 80386. Практически все компьютеры на основе 386-го процессора были оснащены сопроцессором; спроса на компьютеры без такового не было, поскольку незначительное удешевление системы не компенсировало отвратительно медленной работы машины с любыми мало-мальски заметными расчётными задачами. Поэтому при разработке очередного процессора в линейке (486DX) схемы сопроцессора были включены в одну физическую микросхему с основным процессором. Тем не менее, с точки зрения выполняющейся программы арифметический сопроцессор по-прежнему (до сих пор) представляет собой отдельный процессор со своей системой регистров, совсем не похожих на регистры основного процессора, со своими флагами, которые приходится

копировать в основной регистр флагов специальными командами, и со своими своеобразными принципами функционирования.

§ 6.1. Формат чисел с плавающей точкой

Число с плавающей точкой — это особый способ двоичного представления дробного числа, предполагающий отдельное хранение ***мантиссы*** M (двоичной дроби из интервала $1 \leqslant M < 2$) и ***машинного порядка*** P — целого числа, означающего степень двойки, на которую следует умножить мантиссу. Отдельный бит s выделяется под знак числа: если он равен 1 — число считается отрицательным, иначе положительным. Итоговое число, таким образом, вычисляется как $N = (-1)^s M 2^P$. Набор частных соглашений о формате чисел с плавающей точкой, известный как *стандарт IEEE-754*, в настоящее время используется практически всеми процессорами, способными выполнять арифметику с плавающей точкой, и двоичными форматами данных, предполагающими хранение дробных чисел.

Поскольку целая часть мантиссы всегда равна 1, её можно не хранить[1], используя имеющиеся разряды для хранения цифр дробной части. Для хранения машинного порядка в разное время использовались разные способы (знаковое целое с использованием дополнительного кода, отдельный бит для знака порядка и т. п.); стандарт IEEE-754 предполагает хранение машинного порядка в виде *смещённого* беззнакового целого числа: соответствующие разряды рассматриваются как целое число без знака, из которого для получения машинного порядка вычитают некоторую константу, называемую *смещением машинного порядка*.

Стандарт IEEE-754 устанавливает три основных типа чисел с плавающей точкой: число обычной точности, число двойной точности и число повышенной точности[2]. Число обычной точности занимает в памяти 32 бита, из которых один используется для хранения знака числа, восемь — для хранения смещённого машинного порядка (величина смещения — 127) и оставшиеся 23 — для хранения мантиссы. Число двойной точности занимает 64 бита, причём на машинный порядок отводится 11 бит, а на мантиссу — 52, и смещение машинного порядка составляет 1023. Наконец, число повышенной точности занимает 80 бит, из них 15 бит отведено на машинный порядок со смещением 16383, а оставшиеся 64 составляют мантиссу, причём в этом формате присутствует однобитовая целая часть мантиссы (обычно единица).

---

[1] За исключением нескольких особых случаев, о которых речь пойдёт дальше.

[2] Соответствующие англоязычные термины — single precision, double precision и extended precision

Машинный порядок, состоящий из одних нулей или, наоборот, из одних единиц, представляет собой признак особого случая. Порядок, состоящий из одних нулей, означает:

- при мантиссе, состоящей из одних нулей — в зависимости от знакового бита либо ноль, либо «отрицательный ноль» (это различие бывает полезно, если результат очередной операции столь мал по модулю, что его невозможно представить в виде числа с плавающей точкой — тогда мы хотя бы можем сказать, каков был знак результата);
- при мантиссе, содержащей хотя бы одну единицу — ***денормализованное число***, то есть число настолько малое по модулю, что даже при наименьшем возможном значении машинного порядка ни один значащий бит не попал бы в разряды мантиссы.

Порядок, состоящий из одних единиц, может означать следующее:

- при мантиссе, состоящей из одних нулей — «бесконечность» (положительную или отрицательную в зависимости от знакового бита);
- при первом бите мантиссы, установленном в единицу (для 80-битных чисел — при первых двух битах мантиссы, установленных в единицу), а остальных битах мантиссы, установленных в ноль, знаковый бит, равный единице, означает «неопределённость», а знаковый бит, равный нулю — «не-число типа QNAN» (quiet not-a-number); иногда говорят, что неопределённость есть частный случай QNAN;
- при первом бите мантиссы, равном нулю (для 80-битных — при двух первых битах мантиссы, равных 10) и при наличии в остальной мантиссе единичных битов — «не-число типа SNAN»;
- все остальные ситуации (например, мантисса из одних единиц) означают «неподдерживаемое число».

§ 6.2. Устройство арифметического сопроцессора

Арифметический сопроцессор имеет восемь 80-битовых регистров для хранения чисел, которые мы условно обозначим `R0`, `R1`, ..., `R7`; регистры образуют своеобразный *стек*, то есть один из регистров Rn считается вершиной стека и обозначается `ST0`, следующий за ним обозначается `ST1` и т. д., причём считается, что следом за `R7` идёт `R0` (например, если `R7` в настоящий момент обозначен как `ST4`, то роль `ST5` будет играть регистр

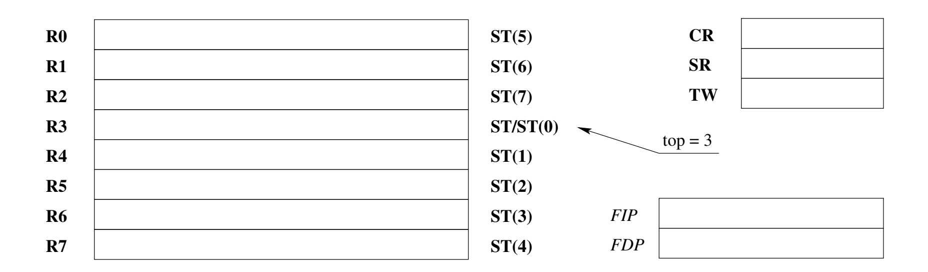


Рис. 6.1. Регистры арифметического сопроцессора

R0, ST6 будет в R1 и т. д.) На рис. 6.1 показана ситуация, когда вершиной стека объявлен регистр R3; роль вершины стека может играть любой из регистров Rn, причём при занесении нового значения в этот стек все значения, которые там уже хранились, остаются на своих местах, а меняется только номер регистра, играющего роль вершины, то есть если в стек, показанный на рисунке, внести новое значение, то роль вершины — ST0 — перейдёт к регистру R2, регистр R3 станет обозначаться ST1, и так далее. При удалении значения из стека происходит обратное действие. Отметим, что к этим регистрам можно обратиться только по их текущему номеру в стеке, то есть по именам ST0, ST1, ..., ST7. Обратиться к ним по их постоянным номерам (R0, R1, ..., R7) нельзя, процессор не даёт такой возможности.

Обозначения ST0, ST1, ..., ST7 соответствуют соглашениям NASM. В других ассемблерах используются другие обозначения; в частности, MASM и некоторые другие ассемблеры обозначают регистры арифметического сопроцессора с использованием круглых скобок: ST(0), ST(1), ..., ST(7), и именно такие обозначения чаще всего встречаются в литературе. Не удивляйтесь этому.

Регистр состояния SR (state register) содержит ряд флагов, описывающих, как следует из названия, состояние арифметического сопроцессора. В частности, биты 13-й, 12-й и 11-й (всего три бита) содержат число от 0 до 7, называемое TOP и показывающее, какой из регистров Rn в настоящий момент считается вершиной стека. Флаги C0 (бит 8), C2 (бит 10) и C3 (бит 14) соответствуют по смыслу флагам центрального процессора CF, PF и ZF. Остальные разряды регистра ST указывают на такие особые ситуации, как переполнение или антипереоплнение стека (SF), потерю точности (P), слишком большой или слишком маленький результат последней операции (O и U), деление на ноль (Z) и др. Регистр управления CR также состоит из отдельных флагов, но, в отличие от регистра статуса, эти флаги обычно устанавливаются программой и предназначены для *управления* сопроцессором, то есть для задания режима его работы. Например, биты 11 и 10 этого регистра задают режим округления результата операции: 00 — к ближайшему числу, 01 — в сторону

уменьшения, `10` — в сторону увеличения, `11` — в сторону нуля (то есть в сторону уменьшения абсолютной величины). Регистр тегов `TW` содержит по два бита для обозначения состояния каждого из регистров `R0`–`R7`: `00` — регистр содержит число, `01` — регистр содержит ноль, `10` — в регистре не-число (NAN, бесконечность или денормализованное число), `11` — регистр пуст. Исходно все восемь регистров помечены как пустые, по мере добавления чисел в стек соответствующие регистры помечаются как заполненные, при извлечении чисел из стека — снова как пустые. Это позволяет отслеживать переполнение и антипереполнение стека — такие ситуации, когда в стек заносится девятое по счёту число (которое некуда поместить), либо, наоборот, делается попытка извлечь число из пустого стека. Эти три регистра мы подробно рассмотрим в § 6.7.3.

Служебные регистры `FIP` и `FDP` предназначены для хранения адреса и операнда последней выполняемой сопроцессором машинной команды и используются операционной системой при анализе причин возникновения ошибочной (исключительной) ситуации.

Мнемонические обозначения всех машинных команд, имеющих отношение к арифметическому сопроцессору, начинаются с буквы `f` от английского *floating* (плавающий; словосочетание «плавающая точка» по-английски звучит как *floating point*). Большинство таких команд не имеет операнда или имеет один операнд, но встречаются и команды с двумя операндами. В качестве операнда могут выступать регистры сопроцессора, обозначаемые `ST`n, либо операнды типа «память». При этом сопроцессор умеет работать с вещественными числами, хранящимися в памяти в любом из трёх форматов, заданных стандартом IEEE-754, что означает, что операнд типа «память» должен быть четырёхбайтным (этот размер можно указать знакомым нам словом `dword`), восьмибайтным или десятибайтным. Для обозначения восьмибайтных операндов ассемблер NASM предусматривает ключевое слово `qword` (от слов *quadro word*, учетверённое слово), а для обозначения десятибайтных — слово `tword` (от *ten word*). Есть и соответствующие псевдокоманды для описания данных (`dq` задаёт восьмибайтное значение, `dt` — десятибайтное), а также для резервирования неинициализированной памяти (`resq` резервирует заданное количество восьмибайтных элементов, `rest` — заданное количество десятибайтных). Сам сопроцессор все действия выполняет с числами повышенной точности, а числа других форматов использует только при загрузке и выгрузке.

§ 6.3. Обмен данными с сопроцессором

Команда `fld` (от слов *float load*), имеющая один операнд, позволяет занести в регистровый стек число из заданного места, в качестве кото-

рого может выступать операнд типа «память» размера dword, qword или tword, либо регистр STn. Например, команда

```
fld st0
```

создаёт копию вершины стека, а команда

```
fld qword [matrix+ecx*8]
```

загружает в стек из массива matrix, состоящего из восьмибайтовых чисел, элемент с номером, хранящимся в регистре ECX. При этом в регистре SR уменьшается значение числа TOP, так что вершина стека сдвигается вверх, старая вершина получает имя ST1 и т. д.

Извлечь результат из сопроцессора (с вершины регистрового стека) можно командами fst и fstp, имеющими один операнд. Чаще всего это операнд типа «память», но можно указать и регистр из стека, например ST6, важно только, что этот регистр должен быть пустым. Основное отличие между этими двумя командами в том, что fst просто читает число, находящееся на вершине стека (т. е. в регистре ST0), тогда как fstp *извлекает* число из стека, помечая ST0 как свободный и увеличивая значение TOP. Команда fst почему-то не умеет работать с 80-битными операндами типа «память», у fstp такого ограничения нет. Отметим ещё один момент: команда

```
fstp st0
```

сначала записывает содержимое ST0 в него же самого, а затем выталкивает ST0 из стека; таким образом, эффект от этой команды состоит в *уничтожении значения на вершине стека.* Так обычно делают в случае, если число, находящееся на вершине стека, в дальнейших вычислениях не нужно.

Часто бывает нужно перевести целое число в формат с плавающей точкой и наоборот. Команда fild позволяет взять из памяти целое число и записать его в стек сопроцессора (естественно, уже в «плавающем» формате). Команда имеет один операнд, обязательно типа «память», размера word, dword или qword (в этом случае имеется в виду восьмибайтное целое). Команды fist и fistp производят обратное действие: берут число, находящееся в ST0, округляют его до целого в соответствии с установленным режимом округления и записывают результат в память по адресу, заданному операндом. По аналогии с командами fst и fstp, команда fst никак не изменяет сам стек, а команда fstp убирает число из стека. Операнд команды fstp может быть размера word, dword или qword, команда fst умеет работать только с word и dword.

Команда fxch позволяет обменять местами содержимое вершины стека (ST0) и любого другого регистра STn, который указывается в качестве

её операнда. Регистры не должны быть пустыми. Чаще всего `fxch` используют, чтобы поменять местами `ST0` и `ST1`, в этом случае операнд можно не указывать.

Сопроцессор поддерживает ряд команд, позволяющих загрузить в стек часто употребляемые константы: `fld1` (загружает 1.0), `fldz` (загружает +0.0), `fldpi` (загружает π), `fldl2e` (загружает $\log_2 e$), `fldl2t` (загружает $\log_2 10$), `fldln2` (загружает $\ln 2$), `fldlg2` (загружает $\lg 2$). Все эти команды не имеют операндов; в результате выполнения каждой из них значение `TOP` уменьшается, и в новом регистре `ST0` оказывается соответствующее значение. От установленного режима округления зависит, в какую сторону будет отличаться загруженное приближённое значение от математического.

§ 6.4. Команды арифметических действий

Простейший способ выполнения четырёх действий арифметики на сопроцессоре — это команды `fadd`, `fsub`, `fsubr`, `fmul`, `fdiv` и `fdivr` с **одним** операндом, в качестве которого может выступать операнд типа «память» размера `dword` или `qword`. Команды `fadd` и `fmul` выполняют соответственно сложение и умножение регистра `ST0` со своим операндом, команда `fsub` вычитает операнд из `ST0`, команда `fdiv` делит `ST0` на свой операнд, `fsubr`, наоборот, вычитает `ST0` из своего операнда, `fdivr` делит свой операнд на `ST0`; результат всех команд записывается обратно в `ST0`. Все шесть команд могут быть использованы и без операндов, в этом случае роль операнда играет `ST1`.

Все перечисленные команды имеют также форму с двумя операндами, при этом в роли обоих операндов могут выступать только регистры STn, причём одним из них обязан быть `ST0` (но он может быть как первым, так и вторым операндом). В этом случае команды выполняют заданное действие над первым и вторым операндами и результат помещают в первый операнд.

Кроме того, все шесть команд имеют ещё и «выталкивающую» форму, которая называется, соответственно, `faddp`, `fsubp`, `fsubrp`, `fmulp`, `fdivp` и `fdivrp`; в этой форме команды имеют всегда два операнда-регистра STn, причём *второй* операнд должен быть `ST0`; после выполнения операции и занесения результата в первый операнд эти команды убирают из стека `ST0`, то есть он помечается как пустой и значение `TOP` увеличивается на единицу; вытесненное из стека число никуда не записывается.

Команды в «выталкивающей» форме можно также записать без операндов, в этом случае в качестве операндов используются `ST1` и `ST0`; действие в этом случае можно описать фразой «взять из стека два операнда, произвести над ними заданное действие, результат положить обратно в

стек». Отметим, что некоторые программисты считают достойными применения только команды в этой форме. Действительно, так можно вычислить любое арифметическое выражение, если только оно не содержит слишком много вложенных скобок (иначе нам не хватит глубины стека). Для этого выражение нужно представить в так называемой польской инверсной записи (ПОЛИЗ), в которой сначала пишутся операнды, потом знак операции; операнды могут быть сколь угодно сложными выражениями, также записанными в ПОЛИЗе. Например, выражение $(x+y)*(1-z)$ в ПОЛИЗе будет записано так: `x y + 1 z - *`. Пусть `x`, `y` и `z` у нас описаны как области памяти (переменные) длины `qword` и содержат числа с плавающей точкой. Тогда для вычисления нашего выражения мы можем просто перевести запись в ПОЛИЗе в запись на языке ассемблера, при этом каждый элемент ПОЛИЗа превратится ровно в одну команду:

```
fld     qword [x]     ; x
fld     qword [y]     ; y
faddp                 ; +
fld1                  ; 1
fld     qword [z]     ; z
fsubp                 ; -
fmulp                 ; *
```

Результат вычисления окажется в ST0. Впрочем, применение других форм арифметических команд способно изрядно укоротить текст программы; как несложно убедиться, следующий фрагмент делает абсолютно то же самое:

```
fld     qword [x]
fadd    qword [y]
fld1
fsub    qword [z]
fmulp
```

Иногда бывают полезны имеющие один операнд команды **fiadd**, **fisub**, **fisubr**, **fimul**, **fidiv** и **fidivr**, выполняющие соответствующее арифметическое действие над ST0 и своим операндом, который должен быть типа «память» размера word или dword и рассматривается как **целое** число.

В заключение разговора о простейшей арифметике упомянем ещё три команды. Команда **fabs** вычисляет модуль ST0, команда **fchs** (от слов *change sign* — сменить знак) меняет знак ST0 на противоположный, команда **frndint** округляет ST0 до целого в соответствии с установленным режимом округления. Результат записывается обратно в ST0 Все три команды имеют только одну форму — без операндов.

Команды **fprem**, **fprem1**, **fscale**, **fxtract** оставляем любознательным читателям для самостоятельного изучения.

§ 6.5. Команды вычисления математических функций

Команды `fsin`, `fcos` и `fsqrt` вычисляют, соответственно, синус, косинус и квадратный корень числа, лежащего в ST0, результат помещается обратно в ST0. Команда `fsincos` чуть сложнее: она извлекает из стека число, вычисляет его синус и косинус и кладёт их в стек, так что синус оказывается в ST1, косинус в ST0, а всего в стеке оказывается на одно число больше, чем было до выполнения команды.

Несколько экзотично ведёт себя команда `fptan`, вычисляющая тангенс. Она берёт аргумент из ST0, вычисляет его тангенс, заносит результат обратно в ST0, но после этого *добавляет в стек ещё число* 1, так что в стеке оказывается на одно число больше, чем до выполнения команды, и при этом в ST0 находится единица, а результат вычисления тангенса находится в ST1. Целью всей этой пляски является упрощение вычисления котангенса: его теперь можно вычислить уже знакомой нам командой `fdivr`; если же котангенс не нужен, избавиться от единицы можно, разделив на неё, то есть командой `fdivp`, или просто выкинуть её из стека командой `fstp st0`.

Команда `fpatan` вычисляет $\mathrm{arctg}\,\frac{y}{x}$, где x — значение в ST0, y — значение в ST1. Эти два числа из стека изымаются, результат записывается в стек, так что в стеке оказывается на одно число меньше, чем было. Знак результата совпадает со знаком y, модуль результата не превосходит π.

Кроме того, сопроцессор предусматривает команды `f2xm1`, `fyl2x` и `fyl2xp`. Команда `f2xm1` вычисляет $2^x - 1$, где x — значение ST0, результат заносит обратно в ST0. Аргумент не должен по модулю превосходить 1, иначе результат неопределён. Команды `fyl2x` и `fyl2xp` вычисляют $y \times \log_2 x$ и $y \times log_2(x+1)$, где x — значение ST0, y — значение ST1; эти значения из стека убираются, а результат добавляется в стек, так что в итоге в стеке остаётся на одно число меньше, чем было, и на вершине находится результат вычисления. При выполнении `fyl2xp1` значение x не должно по модулю превосходить $1 + \frac{\sqrt{2}}{2}$, в противном случае результат неопределён. Читателю предлагается самостоятельно догадаться, для чего нужны эти три команды и как ими пользоваться.

Операнды у всех команд из этого параграфа не предусмотрены.

§ 6.6. Сравнение и обработка его результатов

Общая идея сравнения и действий в зависимости от его результатов для чисел с плавающей точкой такая же, как и для целых: сначала производится сравнение, по итогам которого устанавливаются флаги, а затем используется команда условного перехода в зависимости от состояния флагов. Всё несколько осложняется тем, что у арифметического сопроцессора своя система флагов, причём основной процессор не имеет

команд условного перехода по этим флагам. Поэтому в привычную схему приходится добавить ещё и установку флагов основного процессора в соответствии с текущим состоянием флагов сопроцессора.

Сравнение можно выполнить командами `fcom`, `fcomp` и `fcompp`. Команды `fcom` и `fcomp` имеют один операнд — либо типа «память» размера `dword` или `qword`, либо регистр STn; операнд можно опустить, тогда в его роли выступит ST1. Команды сравнивают ST0 со своим операндом (или с ST1, если операнд не указан. Команда `fcomp` отличается от `fcom` тем, что выталкивает из стека ST0. Команда `fcompp`, не имеющая операндов, сравнивает ST0 с ST1 и выталкивает их оба из стека.

В результате выполнения команд сравнения устанавливаются флаги C3 и C0 в регистре SR (см. стр. 169) следующим образом: при *равенстве* сравниваемых чисел C3 устанавливается в единицу, C0 — сбрасывается в ноль; в противном случае C3 сбрасывается, и если первое из сравниваемых (то есть число, находившееся в регистре ST0) *больше* второго (заданного операндом или регистром ST1), то C0 устанавливается в единицу, если же *меньше* — то сбрасывается. Флаг C3 оказывается, таким образом, по смыслу аналогичным флагу ZF, а флаг C0 — флагу CF (при сравнении беззнаковых целых).

На самом деле команды сравнения устанавливают ещё и флаг C2, причём если всё в порядке — то он сбрасывается в ноль, если же числа *несравнимы* (например, оба числа — «плюс бесконечности», или одно из них — «не-число») и сопроцессор при этом настроен так, чтобы не инициировать прерывания в этих ситуациях — то C2 устанавливается в единицу.

Чтобы результатом сравнения можно было воспользоваться для условного перехода, необходимо скопировать флаги из CR в регистр FLAGS основного процессора. Это делается командами

```
fstsw ax
sahf
```

Первая из них копирует SR в регистр AX, а вторая загружает некоторые (не все!) флаги в FLAGS из AH. В частности, после выполнения этих двух команд значение флага C3 копируется в ZF, а значение C0 — в CF[3], что полностью соответствует нашим потребностям: теперь мы можем воспользоваться для условного перехода любой из команд, предусмотренных для беззнаковых целых чисел: `ja`, `jb`, `jae`, `jbe`, `jna` и т. д. (см. табл. 2.3 на стр. 62). Подчеркнём ещё раз, что использование именно этих команд обусловлено только тем, что результат сравнения оказался во флагах CF и ZF, больше ничего общего между числами с плавающей точкой и беззнаковыми целыми, вообще говоря, нет.

[3]Отметим на всякий случай, что флаг C2 при этом копируется в PF.

Пусть, например, у нас есть переменные a, b и m размера qword, содержащие числа с плавающей точкой, и мы хотим занести в m наименьшее из a и b. Это можно сделать так:

```
        fld qword [b]   ; b на вершину стека (в ST0)
        fld qword [a]   ; теперь a в ST0, b в ST1
        fcom            ; сравниваем их
        fstsw ax        ; копируем флаги в AX
        sahf            ;    и оттуда - в FLAGS
        ja lpa          ; если a>b - прыгаем
        fxcn            ; иначе меняем числа местами
lpa:               ; теперь большее в ST0, меньшее в ST1
        fstp st0        ; ликвидируем ненужное большее
        fstp qword [m]  ; записываем в память меньшее
```

«Ненужное» число можно было бы убрать из стека и иначе. Вместо предпоследней команды можно было бы дать две команды: сначала ffree st0, которая пометит регистр ST0 как свободный, потом fincstp, которая увеличит значение TOP на единицу. Эти команды рассматриваются в §6.7.3.

В ряде случаев могут оказаться полезны также команды ficom и ficomp, всегда имеющие один операнд типа «память» размера word или dword и рассматривающие этот операнд как целое число. В остальном они аналогичны командам fcom и fcomp: первым операндом сравнения выступает ST0, по результатам сравнения устанавливаются флаги C3, C2 и C0. Команда ficomp, в отличие от ficom, выталкивает ST0 из стека. Наконец, команда ftst, не имеющая операндов, сравнивает вершину стека с нулём.

§ 6.7. Управление сопроцессором

§ 6.7.1. Исключительные ситуации и их обработка

В результате выполнения вычислений с плавающей точкой могут возникать ***исключительные ситуации***, что в некоторых случаях свидетельствует об ошибке в программе или входных данных, а в других случаях может отражать вполне штатные особенности хода вычислений. Различают шесть таких ситуаций:

1. Недопустимая операция (Invalid Operation, #I) — попытка использования «не-чисел» в качестве операндов, попытка извлечь квадратный корень или логарифм из отрицательного числа и т. п. Также это может означать ошибку стека: попытку записать новое число в заполненный стек (то есть когда все восемь регистров заняты), либо попытку вытолкнуть число из стека, когда в стеке нет ни одного

числа, либо попытку использовать в качестве операнда регистр, который в настоящее время пуст.

2. Денормализация (Denormalized, `#D`) — попытка выполнения операции над денормализованным числом, либо результат очередной операции столь мал по модулю, что не может быть преставлен иначе как в виде денормализованного числа.

3. Деление на ноль (Zero divider, `#Z`) — попытка деления на ноль.

4. Переполнение (Overflow, `#O`) — результат очередной операции столь велик, что не может быть представлен в виде числа с плавающей точкой имеющихся размеров (частным случаем этой ситуации является перевод числа из внутреннего десятибайтного представления в четырёх- или восьмибайтное представление с помощью, например, команды `fst` в случае, если в новое представление число «не влезает»).

5. Антипереполнение (Underflow, `#U`) — результат столь мал по модулю, что не может быть представлен в виде числа с плавающей точкой нужного размера (в том числе при выполнении команды `fst`, см. выше).

6. Потеря точности (Precision, `#P`) — результат операции не может быть представлен точно имеющимися средствами; в большинстве случаев это абсолютно нормально.

В каждом из регистров `CR` и `SR` младшие шесть бит соответствуют перечисленным ситуациям в том порядке, в котором они перечислены: бит № 0 соответствует недопустимой операции, бит № 1 — денормализации, и т. д.; бит № 5 соответствует потере точности. Кроме того, в регистре `SR` бит № 6 соответствует ошибке стека. При этом биты регистра `CR` *управляют* тем, что процессор должен сделать при возникновении исключительной ситуации. Если соответствующий бит сброшен, то при возникновении исключения будет инициировано *внутреннее прерывание* (см. § 4.2.2). Если же бит установлен, исключительная ситуация считается *замаскированной* и процессор при её возникновении никаких прерываний инициировать не будет; вместо этого он постарается синтезировать, насколько это возможно, релевантный результат (например, при делении на ноль результатом будет «бесконечность» соответствующего знака; при потере точности результат округлится до машинно-представимого числа в соответствии с установленным режимом округления, и т. д.)

При возникновении любой исключительной ситуации сопроцессор устанавливает в единицу соответствующий бит (флаг) в регистре `SR`. Если ситуация не замаскирована, этот бит пригодится операционной системе в обработчике прерывания, чтобы понять, что произошло; если

же ситуация замаскирована и прерывания не произойдёт, установленные флаги можно использовать в программе, чтобы отследить возникшие исключения. Следует учитывать, что эти флаги сами по себе никогда не сбрасываются, их можно сбросить только явно, и это делается командой `fclex`. Команды для взаимодействия с регистрами `CR` и `SR` мы подробно рассмотрим в § 6.7.3.

§ 6.7.2. Параллельное выполнение и команда `wait`

Сопроцессор, являясь логически обособленной частью процессора, не имеет доступа к машинным командам, находящимся в памяти, и не умеет их декодировать; декодирование команд осуществляет основной процессор, он же выдаёт сопроцессору указания к действию. При этом сопроцессор выполняет команды *асинхронно*, то есть основной процессор может продолжать выполнение «своих» команд (таких, чьи имена не начинаются с `F`), не дожидаясь результата работы сопроцессора. С одной стороны, это позволяет повысить эффективность работы программ; с другой стороны, такая параллельная работа может создать определённые проблемы в двух случаях: во-первых, когда последная F-команда записывает что-то в оперативную память, а очередная команда основного процессора должна этот результат использовать; и, во-вторых, когда очередная операция сопроцессора приводит к возникновению исключительной ситуации — при этом дальнейшая работа основной программы может быть бессмысленной, но из-за асинхронного выполнения F-команд прерывание может возникнуть, когда основная программа уже успела выполнить ряд инструкций.

Для синхронизации работы основного процессора с арифметическим сопроцессором используется команда `fwait` или просто `wait` (на самом деле это два обозначения одной и той же машинной команды). Эта команда дожидается завершения всех действий, которые были арифметическому сопроцессору заданы; в том числе, если в результате этих действий было инициировано прерывание, то выполнение команд после `wait` продолжится уже после возврата из прерывания, если, конечно, таковой вообще состоится (обычно в ОС Unix прерывание, инициированное сопроцессором, приводит к отправке сигнала `SIGFPE` текущему процессу, в результате чего процесс завершается).

Интересно, что многие мнемонические обозначения команд сопроцессора на самом деле соответствуют двум машинным командам: сначала идёт команда `wait`, затем — команда, выполняющая нужное действие. Примером такой мнемоники является уже знакомая нам `fstsw`: на самом деле, это две команды — `wait` и `fnstsw`; при необходимости можно использовать `fnstsw` отдельно, без ожидания, но для этого необходимо твёрдо понимать, что́ именно вы делаете. Точно так же устроена команда

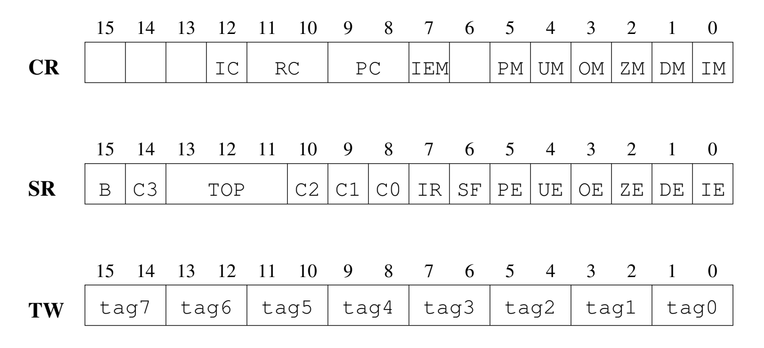


Рис. 6.2. Разряды регистров CR, SR и TW

`fclex` из предыдущего параграфа: это обозначение соответствует машинным командам `wait` и `fnclex`.

§ 6.7.3. Регистры `CR`, `SR` и `TW`

Как уже говорилось, управление режимом работы сопроцессора осуществляется установкой содержимого регистра `CR` (*Control Register*), а по результатам выполнения операций процессор устанавливает содержимое регистра `SR` (*Status Register*), которое можно проанализировать. Наконец, текущее состояние регистров, составляющих стек, отражено в регистре `TW` (*Tag Word*). Назначение разрядов, составляющих регистр управления `CR`, регистр состояния `SR` и регистр меток `TW`, показано на рис. 6.2. Бо́льшая часть этих разрядов нам уже известна; так, младшие шесть бит в регистрах `CR` и `SR` представляют собой соответственно маски и флаги для шести типов исключительных ситуаций (см. § 6.7.1). Биты `IC` и `IEM` регистра `CR` в современных процессорах не используются. Биты `RC` (*Rounding Control*) управляют режимом округления: `00` — к ближайшему числу, `01` — в сторону уменьшения, `10` — в сторону увеличения, `11` — в сторону нуля. Биты `PC` (*Precision Control*) задают точность выполняемых операций: `00` — 32-битные числа, `10` — 64-битные числа, `11` — 80-битные числа (по умолчанию используется именно этот режим, и необходимость его изменить возникает крайне редко).

В регистре `SR` флаги `C3`, `C2` и `C0` обычно используются как признак результата операции сравнения (см. § 6.6); флаг `C1` обычно не используется; флаг `SF` указывает на происшедшую ошибку стека. Флаг `IR` (*Interrupt Request*) указывает на возникновение незамаскированной исключительной ситуации, в результате чего инициировано внутреннее прерывание; увидеть этот флаг установленным можно только в обработчике прерывания внутри операционной системы, так что нас он не касается. Значе-

ние TOP, как уже говорилось, задаёт текущую позицию вершины стека (см. § 6.2). Наконец, бит B (*Busy*) означает, что сопроцессор в настоящий момент занят асинхронным выполнением команды. Надо сказать, что в современных процессорах этот бит тоже невозможно увидеть установленным иначе как в обработчике прерывания.

Регистр TW мы уже рассматривали на стр. 170.

Для работы с регистром CR предусмотрены команды fstcw, fnstcw и fldcw. Команда fstcw, как обычно, означает две машинные инструкции wait и fnstcw. Все три команды имеют один операнд, в качестве которого может выступать только операнд типа «память» размером word. Первые две команды записывают содержимое регистра CR в заданное место в памяти, последняя команда, наоборот, загружает содержимое регистра CR из памяти. Например, следующими командами мы можем установить режим округления «в сторону нуля» вместо используемого по умолчанию режима «к ближайшему»:

```
sub esp, 2          ; выделяем память в стеке
fstcw [esp]         ; получаем в неё содержимое CR
or word [esp], 0000110000000000b
                  ; принудительно устанавливаем биты 11 и 10
fldcw [esp]         ; загружаем полученное обратно в CR
add esp, 2          ; освобождаем память
```

Содержимое регистра SR можно получить уже знакомой нам командой fstsw, операнд которой может быть либо регистром AX (и больше никаким), либо типа «память» размером word. Имеется также команда fnstsw, причём fstsw представляет собой обозначение для двух машинных инструкций wait и fnstsw. Отметим, что обратная операция (загрузка значения) для SR не предусмотрена, что вполне логично: этот регистр нужен, чтобы анализировать происходящее. Тем не менее, некоторые команды воздействуют на этот регистр напрямую. Так, значение TOP можно увеличить на единицу командой fincstp и уменьшить на единицу командой fdecstp (обе команды не имеют операндов). Использовать эти команды следует осторожно, поскольку статус «занятости» регистров стека они не меняют; иначе говоря, fdecstp приводит к тому, что регистром ST0 становится «пустой» регистр, а fincstp приводит к тому, что ST7 оказывается «занят» (поскольку это бывший ST0). Ещё одно активное действие с регистром SR, которое может выполнить программист — это очистка флагов исключительных ситуаций. Такая очистка производится командами fclex (*Clear Exceptions*) и fnclex, которые мы уже упоминали в предыдущем параграфе.

Перед командой fldcw рекомендуется всегда выполнять команду fclex, иначе может случиться так, что запись регистра CR «демаскирует» какое-нибудь из исключений, флаг которого уже взведён, в результате чего произойдёт прерывание.

Регистр TW не может быть напрямую ни считан, ни записан, но одна команда, напрямую воздействующая на него, всё же есть. Она называется ffree, имеет один операнд — регистр STn, а её действие — пометить заданный регистр как «свободный» (или «пустой»). В частности, следующие команды убирают число с вершины стека «в никуда»:

```
ffree st0
fincstp
```

§ 6.7.4. Инициализация, сохранение и восстановление

Если на момент начала вычислений вам не известно (или вызывает сомнения) состояние арифметического сопроцессора, но при этом вы точно знаете, что никакой полезной для вас информации его регистры не содержат, можно привести его «в исходное состояние» с помощью команды finit или fninit (finit представляет собой обозначение для wait fninit, см. § 6.7.2). При этом в регистр CR заносится значение 037Fh (округление в ближнюю сторону, наибольшая возможная точность, все исключения замаскированы); регистр SR обнуляется, что означает TOP=0, все флаги сброшены, включая флаги исключительных ситуаций; регистры FIP, FDP, TW также обнуляются; регистры, составляющие стек, никак не изменяются, но поскольку TW обнулён, все они считаются свободными (не содержащими чисел).

С помощью команды fsave можно сохранить всё состояние сопроцессора, то есть содержимое всех его регистров, в области памяти, чтобы потом восстановить его. Это полезно, если нужно временно прекратить некий вычислительный процесс, выполнить какие-то вспомогательные вычисления, затем вернуться к отложенному процессу вычислений. Для сохранения вам потребуется область памяти длиной 108 байт; команда fsave имеет один операнд, это операнд типа «память», причём указывать его размер не нужно. Мнемоника fsave на самом деле обозначает две машинные команды — wait и fnsave. После сохранения состояния в памяти сопроцессор приводится «в исходное состояние» точно так же, как при команде finit (см. выше), так что после fsave отдельно давать команду finit не нужно. Восстановить сохранённое ранее состояние сопроцессора можно командой frstor; как и fsave, эта команда имеет один операнд типа «память», для которого не нужно указывать размер, поскольку используется область памяти размером 108 байт.

Иногда возникает потребность сохранить или восстановить только вспомогательные регистры сопроцессора. Это делается командами fsetenv, fnsetenv и fldenv с использованием области памяти длиной 28 байт; подробное описание этих команд оставляем за рамками пособия.

В завершение разговора о сопроцессоре упомянем команду fnop. Как можно догадаться, это очень важная команда: она *не делает ничего.*

Приложение: текст файла stud_io.inc

Версия для ОС Linux

```
;; system dependend part ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; generic 3-param syscall
%macro _syscall_3 4
        push edx
        push ecx
        push ebx
        push %1
        push %2
        push %3
        push %4
        pop edx
        pop ecx
        pop ebx
        pop eax
        int 0x80
        pop ebx
        pop ecx
        pop edx
%endmacro
; syscall_exit is the only syscall we use that has 1 parameter
%macro _syscall_exit 1
        mov ebx, %1     ; exit code
        mov eax, 1      ; 1 = sys_exit
        int 0x80
%endmacro
;; system dependent part ends here ;;;;;;;;;;;;;;;;;;;;;;;;;;;

; %1: descriptor   %2: buffer addr   %3: buffer length
; output: eax: read bytes
%macro _syscall_read 3
        _syscall_3 3,%1,%2,%3
%endmacro

; %1: descriptor   %2: buffer addr   %3: buffer length
; output: eax: written bytes
%macro _syscall_write 3
        _syscall_3 4,%1,%2,%3
%endmacro

%macro  PRINT 1
        pusha
        pushf
        jmp %%astr
```

```
%%str   db      %1, 0
%%strln equ     $-%%str
%%astr: _syscall_write 1, %%str, %%strln
        popf
        popa
%endmacro

%macro  PUTCHAR 1
        pusha
        pushf
%ifstr %1
        mov     al, %1
%elifnum %1
        mov     al, %1
%elifidni %1,al
        nop
%elifidni %1,ah
        mov     al, ah
%elifidni %1,bl
        mov     al, bl
%elifidni %1,bh
        mov     al, bh
%elifidni %1,cl
        mov     al, cl
%elifidni %1,ch
        mov     al, ch
%elifidni %1,dl
        mov     al, dl
%elifidni %1,dh
        mov     al, dh
%else
        mov     al, %1  ; memory location such as [var]
%endif
        sub     esp, 2  ; reserve memory for buffer
        mov     edi, esp
        mov     [edi], al
        _syscall_write 1, edi, 1
        add     esp, 2
        popf
        popa
%endmacro

%macro  GETCHAR 0
        pushf
        push    edi
        sub     esp, 2
        mov     edi, esp
```

```
        _syscall_read 0, edi, 1
        cmp     eax, 1
        jne     %%eof_reached
        xor     eax,eax
        mov     al, [edi]
        jmp     %%gcquit
%%eof_reached:
        xor     eax, eax
        not     eax             ; eax := -1
%%gcquit:
        add     esp, 2
        pop     edi
        popf
%endmacro

%macro FINISH 0-1 0
        _syscall_exit %1
%endmacro
```

Версия для FreeBSD

Эта версия отличается от предыдущей только определением макросов `_syscall3` и `_syscall_exit`, поэтому целиком мы её не приводим. Чтобы получить рабочий файл для ОС FreeBSD, возьмите вышеприведённый текст для ОС Linux и замените определения этих макросов на следующие:

```
;; freebsd-specific things ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro _syscall_3 4
        push    %4
        push    %3
        push    %2
        mov     eax, %1
        push    eax
        int     0x80
        jnc     %%ok
        neg     eax
%%ok:   add     esp, 16
%endmacro

%macro _syscall_exit 1
        push    %1      ; exit code
        mov     eax, 1  ; 1 = sys_exit
        push    eax
        int     0x80
        ; no cleanup - this will never return anyway
%endmacro
;; system dependent part ends here ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Литература

[1] Э. Танненбаум. Архитектура компьютера. 4-е издание. СПб.: Питер, 2003.

[2] Зубков С. В. Assembler для DOS, Windows и UNIX. М.:ДМК, 2006.

[3] Баурн С. Операционная система UNIX. М.:Мир, 1986.

[4] Робачевский А. М. Операционная система UNIX. Изд-во «BHV», Санкт-Петербург, 1997.

[5] The Netwide Assembler: NASM. `http://www.nasm.us/doc/` *Имеется русский перевод, выполненный AsmOs group; см. например, `http://opslab.org.ru/nasm`*

[6] Raymond Filiatreault. Simply FPU (an FPU tutorial). 2003. `http://www.ray.masmcode.com/fpu.html`

Домашняя страница этой книги в сети Интернет расположена по адресу
`http://www.stolyarov.info/books/asm_unix`
Здесь вы можете получить тексты примеров программ, приведённых в этой книге, а также электронную версию самой книги.

Оглавление